TinySec: A Link Layer Security Architecture for Wireless Sensor Networks

Chris Karlof
ckarlof@cs.berkeley.edu
UC Berkeley

Naveen Sastry
nks@cs.berkeley.edu
UC Berkeley

David Wagner
daw@cs.berkeley.edu
UC Berkeley

ABSTRACT

We introduce TinySec, the first fully-implemented link layer security architecture for wireless sensor networks. In our design, we leverage recent lessons learned from design vulnerabilities in security protocols for other wireless networks such as 802.11b and GSM. Conventional security protocols tend to be conservative in their security guarantees, typically adding 16–32 bytes of overhead. With small memories, weak processors, limited energy, and 30 byte packets, sensor networks cannot afford this luxury. TinySec addresses these extreme resource constraints with careful design; we explore the tradeoffs among different cryptographic primitives and use the inherent sensor network limitations to our advantage when choosing parameters to find a sweet spot for security, packet overhead, and resource requirements. TinySec is portable to a variety of hardware and radio platforms. Our experimental results on a 36 node distributed sensor network application clearly demonstrate that software based link layer protocols are feasible and efficient, adding less than 10% energy, latency, and bandwidth overhead.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection Cryptographic controls

General Terms

Security, Design

Keywords

Sensor Network Security, Link Layer Security

1. INTRODUCTION & MOTIVATION

There is considerable excitement about new applications enabled by sensor networks, and we are on the cusp of a broader deployment of these technologies. However, one challenge that faces us is the question of how to secure sensor networks: without adequate security, widespread deployment could be curtailed.

We have taken up this challenge and introduce TinySec, a lightweight, generic security package that developers can easily integrate into sensor network applications. We foresee TinySec will cover the basic security needs of all but the most security critical applications. As a part of this, we were motivated by an observation about 802.11 wireless networks: several studies report that 50-80% of all 802.11 wireless networks operate in the clear, without any cryptographic protection whatsoever [24, 36, 37, 45]. To achieve high deployment rates in sensor networks, we believe that a security system must be easy to use and minimally impact performance. Failure to meet either requirement creates a justifiable reason for developers to leave out security.

We base the design of TinySec on existing security primitives that other researchers have proven to be secure. Using these primitives, we design a lightweight and efficient link-layer security protocol that is tailored to sensor networks. We describe a complete solution, defining packet formats and application interfaces, and provide a detailed performance characterization. Previous work, such as SNEP [33], analyzed aspects of the design space. Many of their design choices are sound, but further experience with sensor networks lead us to reevaluate their work as researchers gain more understanding of the limitations and capabilities of the devices. Correspondingly, we note that much of the value of a link-layer security system comes from the higher level algorithms with which it is paired. We designed TinySec with this in mind and built TinySec as a research platform for use in testing and evaluating higher level security packages.

One of the major barriers to deploying security on sensor networks is that current sensor devices have limited computation and communication capabilities. Since cryptography is not free, these performance constraints pose a non-trivial challenge for any system that would incorporate cryptography into sensor networks. We expect that people will use Moore's law to drive down the cost of these devices and not to increase their performance capabilities. But with a careful analysis, we can use the inherent limitations to our advantage. For example, the bandwidth of the wireless channel used in sensor networks is significantly less than that of conventional networks. This implies that even a powerful adversary is limited in how many packets per second she can inject or eavesdrop on. Designing protocols that rely on properties such as these is one strategy we take in reducing overhead. Our design choices are driven by sensor network capabilities and realities; this ultimately separates TinySec from other low overhead security protocols.

Before this work, an interesting open problem was whether software cryptography could achieve acceptable performance on typical sensor platforms, or whether hardware assistance would be needed. Many previous systems (e.g., GSM, Bluetooth, 802.15.4) took the stance that hardware is needed. In contrast, we show that, with sufficient engineering effort, it is possible to encrypt and authenticate all communications entirely in software, without special hardware, and without major performance degradation.

The main contributions of this paper are:

- We introduce TinySec, the first fully-implemented protocol for link-layer cryptography in sensor networks. We have incorporated our implementation of TinySec into the official TinyOS release.

- We explore some of the tradeoffs between performance, transparency, and cryptographic security, and we propose a design that meets the needs of applications in the sensor network space.

- We measure the bandwidth, latency, and energy costs of our implementation of TinySec and show that they are minimal for sensor network applications. This demonstrates for the first time that it is feasible to implement acceptable cryptographic protection for sensor networks entirely in software.

- TinySec is a research platform that is easily extensible and has been incorporated into higher level protocols. We have evidence of several sensor network security projects using TinySec in their research.

2. SENSOR NETWORKS

We use the term *sensor network* to refer to a heterogeneous system combining tiny sensors and actuators with general-purpose computing elements. We envision sensor networks will consist of hundreds or thousands of low-power, low-cost wireless nodes deployed en masse to monitor and affect the environment. Applications include habitat monitoring [5, 31, 39], burglar alarms, inventory control, medical monitoring and emergency response [44], and battlefield management [17].

A representative sensor node is the Mica2 [23], a several cubic inch sensor/actuator unit with a CPU, radio, power source, and optional sensing elements. The processor is a 8 MHz 8-bit Atmel ATMEGA128L CPU with 128 kB of instruction memory, 4 kB of RAM for data, and 512 kB of flash memory. It features a low-powered radio from Chipcon, delivering up to 19.2 kbps application bandwidth on a single shared channel and with a range of up to around hundred meters. At full power, the Mica2 sensor node can run for only two weeks or so before exhausting its batteries.

Mica2 sensor nodes run TinyOS [23], an event-driven operating system for networked applications in wireless embedded systems. The memory footprint of TinyOS is small, with the core components requiring only 400 bytes of data and instruction memory. TinyOS supports other hardware platforms as well.

It is clear that we must discard many preconceptions about network security: sensor networks differ from other distributed systems in important ways. These devices have very little computational power; even efficient public-key cryptography and fast symmetric ciphers must be used with care. There is considerable pressure to ensure that our security protocols use a minimal amount of the limited RAM. Additionally, communication bandwidth is extremely dear: each bit transmitted consumes about as much power as executing 800–1000 instructions [23], and as a consequence, any message expansion caused by security mechanisms comes at significant cost. Energy is the scarcest resource of all: each milliamp consumed is one milliamp closer to death, and as a result, nearly every aspect of sensor networks must be designed with power in mind.

2.1 Security risks and threat models in sensor networks

Because sensor networks use wireless communication, they are vulnerable to attacks which are more difficult to launch in the wired domain. Many wired networks benefit from their inherent physical security properties. It is unlikely that an adversary will dig up the Internet backbone and splice into the line. However, wireless communications are difficult to protect; they are by nature a broadcast medium. In a broadcast medium, adversaries can easily eavesdrop on, intercept, inject, and alter transmitted data. In addition, adversaries are not restricted to using sensor network hardware. They can interact with the network from a distance by using expensive radio transceivers and powerful workstations.

Sensor networks are vulnerable to resource consumption attacks. Adversaries can repeatedly send packets to drain the nodes' batteries and waste network bandwidth. Since sensor networks will be deployed in a variety of physically insecure environments, adversary can steal nodes, recover their cryptographic material, and pose as authorized nodes in the network. However, we do not address these threats. Our focus is on guaranteeing message authenticity, integrity, and confidentiality. We do not address resource consumption attacks, physical tamper resistance, or node capture attacks.

2.2 Motivation for link-layer security in sensor networks

In conventional networks, message authenticity, integrity, and confidentiality are usually achieved by an end-to-end security mechanism such as SSH [47], SSL [3], or IPSec [4] because the dominant traffic pattern is end-to-end communication; intermediate routers only need to view message headers and it is neither necessary nor desirable for them to have access to message bodies.

This is not the case in sensor networks. The dominant traffic pattern in sensor networks is many-to-one, with many sensor nodes communicating sensor readings or network events over a multihop topology to a central base station. However, neighboring nodes in sensor networks often witness the same or correlated environmental events, and if each node sends a packet to the base station in response, precious energy and bandwidth are wasted. To prune these redundant messages to reduce traffic and save energy, sensor networks use in-network processing such as aggregation and duplicate elimination [29, 30]. Since in-network processing requires intermediate nodes to access, modify, and suppress the contents of messages, it is unlikely we can use end-to-end security mechanisms between each sensor node and the base station to guarantee the authenticity, integrity, and confidentiality of these messages.

End-to-end security mechanisms are also vulnerable to certain denial of service attacks. If message integrity is only checked at the final destination, the network may route packets injected by an adversary many hops before they are detected. This kind of attack will waste precious energy and bandwidth. A link-layer security architecture can detect unauthorized packets when they are first injected into the network. Link-layer security mechanisms have been proposed for wired networks to resist similar denial of service attacks [22].

For the above reasons, we decided on a link-layer security architecture for TinySec. Link-layer security mechanisms guarantee the authenticity, integrity, and confidentiality of messages between neighboring nodes, while permitting in-network processing. Despite the problems enumerated above, end-to-end security mechanisms can still useful in sensor networks and complement TinySec.[1]

3. DESIGN GOALS

We have three goals for a link layer security mechanism in sensor networks: security, performance, and usability.

3.1 Security goals

A link layer security protocol should satisfy three basic security properties: *access control*, *message integrity*, and *message confidentiality*.

Access control and message integrity. Access control means the link layer protocol should prevent unauthorized parties from participating in the network. Legitimate nodes should be able to detect messages from unauthorized nodes and reject them. Closely related to message authenticity is message integrity: if an adversary modifies a message from an authorized sender while the message is in transit, the receiver should be able to detect this tampering. We provide message authentication and integrity by including a *message authentication code* with each packet. We discuss message authentication codes in more detail in Section 4.

Confidentiality. Confidentiality means keeping information secret from unauthorized parties. It is typically achieved with encryption. Preferably, an encryption scheme should not only prevent message recovery, but also prevent adversaries from learning even partial information about the messages that have been encrypted. This strong property is known as *semantic security* [8]. Semantic security implies adversaries should have no better than a 50% chance in correctly answering any yes or no question about an encrypted message. We discuss mechanisms for achieving semantic security in more detail in Section 4.

Explicit omission: Replay protection. An adversary that eavesdrops on a legitimate message sent between two authorized nodes and replays it at some later time engages in a *replay attack*. Since the message originated from an authorized sender, the same receiver will accept it again. Replay protection is a difficult problem when there is a limited amount of state that each recipient keeps.

[1]By removing the link-layer protocol headers, the TinySec packet format could be used in an end-to-end security protocol as well. We do not address this modification in this paper.

A common defense is to include a monotonically increasing counter with every message and reject messages with old counter values. With this policy, every recipient must maintain a table of the last value from every sender it receives. However, for RAM-constrained sensor nodes, this defense becomes problematic for even modestly sized networks. Assuming nodes devote only a small fraction of their RAM for this neighbor table, an adversary replaying broadcast messages from many different senders can fill up the table. At this point, the recipient has one of two options: ignore any messages from senders not in its neighbor table, or purge entries from the table. Neither is acceptable; the first creates a DoS attack and the second permits replay attacks. This is a realistic concern. If each counter requires 4 bytes and there is only 100 bytes of RAM available for the neighbor table (2.5% of the total RAM), networks larger than 25 nodes will be vulnerable.

However, the application layer may be better equipped to manage the replay table if it expects certain communication patterns or has information about the network topology. This type of information is typically not available at the link layer. For example, if the physical topology implies that only nodes 1, 2, 3, 4 should be able to communicate with node 5, an application running on node 5 can efficiently manage the neighbor table by only keeping replay counter entries for these four nodes. For this reason, we believe replay protection belongs not in the link-layer, but rather in higher layers of the protocol stack. By using information about the network's topology and communication patterns, the application and routing layers can properly and efficiently manage a limited amount of memory devoted to replay detection.

3.2 Performance

A system using cryptography will incur increased overhead in the length of messages sent as well as in extra demands on the processor and RAM. The increased message length can decrease message throughput and increase latency, but more importantly for sensor networks, it will also increase power consumption. We would like to impose only a modest increase in power consumption when using TinySec and achieve comparable channel utilization and latency.

Due to the extreme resource limitations in sensor networks, it is important to carefully tune the strength of the security mechanisms in a way that provides reasonable protection while limiting overhead. This is in sharp contrast to conventional network security where the difference between 8 or 16 bytes of overhead is often inconsequential. Cryptography designed for conventional networks is conservative because it can afford to be. In sensor networks, 8 bytes is nearly 25% of the total packet size, and overly conservative choices of security parameters will consume resources too quickly.

3.3 Ease of use

Security platform. We expect higher level security protocols will rely on the link-layer security architecture as a primitive. For example, key distribution protocols, some of which utilize public key cryptography, could use TinySec to create secure pairwise communication between neighboring nodes. To reduce the effort in implementing these protocols, Tiny-

Sec must provide the right set of interfaces to facilitate their development.

Transparency. A major challenge in deploying security mechanisms is the difficulty in properly using and implementing them. Frequently, application programmers are unsure of appropriate security parameters. Also, if the secure communication mechanism requires different APIs than the standard mechanism, then migrating legacy applications will be difficult. Of course, if it is not easy to enable the security features, many users and programmers will disable it and continue to operate insecurely.

To alleviate these problems, an important design goal of TinySec is that it should be transparent to applications running on TinyOS. To achieve this goal, we structured TinySec as a link-layer security protocol. We believe that transparency will play a crucial role in enabling widespread deployment of TinySec and other security mechanisms.

At the same time, we try to make it easy for security-aware applications to customize the level of security that TinySec provides: application programmers should be able to adjust the security performance tradeoffs if they have a greater understanding of their application's security needs.

Portability. An additional goal of TinySec is that it should be portable. TinyOS runs on a host of different platforms, including processors manufactured by Texas Instruments, Atmel, Intel x86, and StrongArm. TinyOS also supports two radio architectures: the Chipcon CC1000 and the RFM TR1000. A radio stack bridges these two pieces of hardware. A link layer security architecture should fit into the radio stack so that porting the radio stack from one platform to another is a simple job.

4. SECURITY PRIMITIVES

In this section, we give background on some well-studied cryptographic primitives commonly used to achieve our security goals. We apply these primitives in TinySec.

Message authentication codes (MACs). A common solution for achieving message authenticity and integrity is to use a *message authentication code* (MAC).[2] A MAC can be viewed as a cryptographically secure checksum of a message. Computing a MAC requires authorized senders and receivers to share a secret key, and this key is part of the input to a MAC computation. The sender computes a MAC over the packet with the secret key and includes the MAC with the packet. A receiver sharing the same secret key recomputes the MAC and compares it with the received MAC value. If they are equal, the receiver accepts the packet and rejects it otherwise. MACs must be hard to forge without the secret key. This implies if an adversary alters a valid message or injects a bogus message, she cannot compute the corresponding MAC value, and authorized receivers will reject these messages.

[2]There is an unfortunate name collision between the cryptographic and networking community. We will refer to the acronym "MAC" only in the cryptographic sense and use "media access control" to refer to protocols governing access to channel.

Initialization vectors (IVs). Recall we want our encryption mechanism to achieve semantic security (Section 3.1). One implication of semantic security is that encrypting the same plaintext two times should give two different ciphertexts. A common technique for achieving semantic security is to use a unique initialization vector (IV) for each invocation of the encryption algorithm. An IV can be thought of as a side input to the encryption algorithm. The main purpose of IVs is to add variation to the encryption process when there is little variation in the set of messages. Since the receiver must use the IV to decrypt messages, the security of most encryption schemes do not rely on IVs being secret. IVs are typically sent in the clear and are included in the same packet with the encrypted data. We discuss IVs further in Section 5, including their necessary length and how to generate them.

It might seem that given the resource constraints in sensor networks, we could give up semantic security to eliminate the additional packet overhead required by an IV. However, semantic security is almost always necessary and desirable, even in resource constrained environments. Consider application messages with low entropy, such as YES or NO messages that are sent periodically to report environmental events such as movement. Without using an IV, all encryptions of YES messages are identical. Once an adversary determines what a YES message looks like, confidentiality is lost; the adversary can determine the contents of every YES/NO message by simply looking at its encryption.

5. DESIGN OF TINYSEC

5.1 Existing schemes are inadequate

Using cryptography to secure an untrusted channel has been well-studied in the literature, and there are a plethora of existing schemes that try to achieve this goal. In the networking community, protocols such as IPSec, SSL/TLS, and SSH all do a satisfactory job of securing Internet communications. However, these protocols are too heavy-weight for use in sensor networks. Their packet formats add many bytes of overhead, and they were not designed to run on computationally-constrained devices.

The wireless, cellular telephony, and ad-hoc networking communities have developed schemes closer to our needs, but even there, the existing designs have serious limitations. The closest previous work is SNEP [33], which specifically targets sensor networks, but SNEP was unfortunately neither fully specified nor fully implemented. Refer to Section 10 for further discussion of these wireless security mechanisms.

The conclusion is that existing schemes are either insecure or too resource intensive for use in sensor networks, and we must design a new scheme.

5.2 TinySec design

TinySec supports two different security options: authenticated encryption (TinySec-AE) and authentication only (TinySec-Auth). With authenticated encryption, TinySec encrypts the data payload and authenticates the packet with a MAC. The MAC is computed over the encrypted data and the packet header. In authentication only mode, TinySec authenticates the entire packet with a MAC, but the data payload is not encrypted.

5.2.1 *Encryption*

Using semantically secure encryption typically requires two design decisions: selecting an encryption scheme and specifying the IV format. In our design of TinySec, we use a specially formatted 8 byte IV, and we use cipher block chaining (CBC) [8]. In this section, we introduce the structure of our IV format and argue why CBC is the most appropriate encryption scheme for sensor networks.

TinySec IV format. Recall that our goal is to see how much we can reduce the cost of security. The length of our IV, and the way we generate IVs, can have a dramatic effect on both security and on performance. If the IV is too long, we will add unnecessary bits to the packet, which translates to a significant cost in overall throughput and in energy drain. At the same time, if the IV is too short, we run the risk that the IV will repeat, and then our security warranty is void.

How long is long enough? By the pigeonhole principle, a n-bit IV is guaranteed to repeat after $2^n + 1$ packets are sent, no matter how we choose the IV. If we use a n-bit counter, repetitions will not occur before that point. However, for some IV generation strategies, repetitions may occur earlier. If we choose each IV as a random n-bit value, then by the birthday paradox, we expect (probabilistically) to see the first repetition after roughly $2^{n/2}$ packets have been sent. Therefore, we use a counter in our IV, and we transmit it in the packet so that the receiver can learn the value of the counter.

The structure of the IV is $dst||AM||\ell||src||ctr$, where dst is the destination address of the receiver, AM is the active message (AM) handler type, ℓ is the length of the data payload, src is the source address of the sender, and ctr is a 16 bit counter. The counter starts at 0, and the sender increases it by 1 after each message sent. We analyze the security of this construction in Section 6.2.

Encryption schemes. In this section we argue why CBC is the most appropriate encryption scheme for sensor networks. Symmetric key encryption schemes generally fall into two categories: stream ciphers and modes of operation using block ciphers.

A stream cipher (typically) uses a key K and IV as a seed and stretches it into a large pseudorandom keystream $G_K(IV)$. The keystream is then XORed against the message: $C = (IV, G_K(IV) \oplus P)$. The fastest stream ciphers are faster than the fastest block ciphers [41], which might make them look tempting in a resource-constrained environment. However, stream ciphers have a devastating failure mode: if the same IV is ever used to encrypt two different packets, then it is often possible to recover both plaintexts.[3]

Guaranteeing that IVs are never reused requires IVs to be fairly long, say, at least 8 bytes. Since one of our goals is to minimize packet overhead, we believe adding 8 additional bytes to a 30-byte packet is unacceptable. The alternative is require shorter IVs and accept that IV reuse will occur. Therefore, we were guided by the following principle: "Use an encryption scheme that is as robust as possible in the presence of repeated IVs." Stream ciphers clearly violate this principle, so the only alternative is to use a mode of operation based on a block cipher [11].

A block cipher is a keyed pseudorandom permutation over small bit strings, typically 8 or 16 bytes. Examples of block ciphers include DES, AES, RC5, and Skipjack. Since we usually want to encrypt and authenticate messages longer than 8 or 16 bytes, block ciphers require a *mode of operation* to encrypt longer messages. For a k byte block cipher, a mode of operation typically breaks a message into segments of k bytes and uses the block cipher in a special way to encrypt the message block by block.

Using a block cipher for encryption has an additional advantage. Since the most efficient message authentication code (MAC) algorithms use a block cipher, the nodes will need to implement a block cipher in any event. Using this block cipher for encryption as well conserves code space.

If we use block ciphers for encryption, we must choose a mode of operation. One natural choice is counter (CTR) mode [8]; however, CTR mode is a stream cipher mode of operation, and shares all the problems as any other stream cipher. Therefore, we rejected CTR mode.

Another natural choice is cipher block chaining (CBC) mode [8]. CBC mode is better: it degrades more gracefully in the presence of repeated IVs. In particular, if we encrypt two plaintexts P, P' with the same IV under CBC mode, then the ciphertexts will leak the length (in blocks) of the longest shared prefix of P and P', and nothing more. For instance, if the first block of P is different from the first block of P', as will typically be the case, then the cryptanalyst learns nothing apart from this fact. Consequently, CBC leaks only a small amount of information in the presence of repeated IVs, a significant improvement over a stream cipher. CBC mode is provably secure when IVs do not repeat [8].

CBC mode was designed to be used with a random IV, and CBC mode has a separate leakage issue when used with a counter as IV. Suppose we encrypt two plaintexts P, P' under IV, IV', respectively. If $P_1 \oplus IV = P'_1 \oplus IV'$ (where P_1 denotes the first block of P, etc.), then the first block of ciphertexts will be equal, and this discloses the value $P_1 \oplus P'_1$. In some cases, this can leak partial information about plaintexts. For instance, suppose the IV is a counter, and let IV, IV' be two consecutive IVs. We will often have $IV' = IV \oplus 1$. If the plaintexts occasionally satisfy the same pattern, i.e., $P' = P \oplus 1$, then we will have occasional leakage. This is undesirable.

Fortunately, there is a simple fix that allows CBC mode to be used with any non-repeating IV. The fix is to pre-encrypt the IV, and we reject standard CBC mode in favor of this variant.

Naively using CBC mode for encryption with a 8-byte block cipher results in ciphertexts which are multiples of 8 bytes. This may result in message expansion, which increases power consumption. We use a technique known as ciphertext stealing [35] to ensure the ciphertext is the same length as the underlying plaintext. Encrypting data payloads of less than 8 bytes will produce a ciphertext of 8 bytes because ciphertext stealing requires at least one block of ciphertext. However, the fixed overhead of sending a message (turning on the radio, acquiring the channel, and sending the start symbol) generally discourages short messages.

[3]Given $C = (IV, G_K(IV) \oplus P)$ and $C' = (IV, G_K(IV) \oplus P')$, one can recover $P \oplus P'$, which is a lot of information about P and P'. When plaintexts have sufficient redundancy, one can often recover most or all of P and P' from $P \oplus P'$ [15].

Block cipher choice. Conventional wisdom says when a block cipher is needed, choose either AES or Triple-DES. However, Triple-DES is too slow for software implementation in embedded microcontrollers. Our initial experiments showed that AES was quite slow, too. Therefore, we rejected AES.[4]

We surveyed other block ciphers to find one that is well-suited for sensor networks. We found RC5 and Skipjack to most appropriate for software implementation on embedded microcontrollers. We discuss the performance of these ciphers in Section 9.1. Although RC5 is slightly faster, it is patented. Also, for good performance, RC5 requires the key schedule to be precomputed, which uses 104 extra bytes of RAM per key. Because of these drawbacks, the default block cipher in TinySec is Skipjack.

5.2.2 Message integrity: The need for a MAC

One might ask if encryption is enough; do we need an authentication mechanism when messages are encrypted? History has proven that using encryption without authentication is insecure [10, 12, 27]. For example, flipping bits in unauthenticated encrypted messages can cause predictable changes in the plaintext [12], and without an authentication mechanism to guarantee integrity, receivers are unable to detect the changes. Unauthenticated messages are also vulnerable to cut-and-paste attacks [10]. In a cut-and-paste attack, an adversary breaks apart an unauthenticated encrypted message and constructs another message which decrypts to something meaningful. For example, if all the authorized nodes share a single key, an adversary can extract the encrypted data payload from a message to one node and send it to different node. Since the encrypted payload is unaltered, the second node will successfully decrypt and accept the message.

To address these vulnerabilities, TinySec always authenticates messages, but encryption is optional. Message confidentiality is only necessary when there is something to keep secret. Consider a burglar alarm. The actual contents of an alarm message could be empty; receiving an alarm message signals an intrusion. Encryption is unnecessary and only increases latency, computation, and power consumption. However, most all applications require packet authenticity, meaning authorized nodes will not accept invalid messages injected by an adversary. In our burglar alarm example, this means adversaries cannot trigger false alarms.

TinySec uses a cipher block chaining construction, CBC-MAC [9], for computing and verifying MACs. CBC-MAC is efficient and fast, and the fact that it relies on a block cipher as well minimizes the number of cryptographic primitives we must implement in the limited memory we have available. CBC-MAC is provably secure [9], however the standard CBC-MAC construction is not secure for variably sized messages. Adversaries can forge a MAC for certain messages. Bellare, Kilian, and Rogaway suggest three alternatives for generating MACs for variable sized messages [9]. The variant we use XORs the encryption of the message length with the first plaintext block.

[4]We have since been informed that AES can be implemented efficiently on our platform, with performance not much worse than RC5 and Skipjack. AES does have the small disadvantage that its block length is longer. However, in retrospect, AES might be a perfectly suitable replacement.

5.3 Packet format

We based TinySec's packet format on the current packet format in TinyOS. We show the differences between TinySec packets and TinyOS packets in Figure 1. The common fields are destination address, active message (AM) type, and length. Active message types are similar to port numbers in TCP/IP. The AM type specifies the appropriate handler function to extract and interpret the message on the receiver. These fields are unencrypted because the benefits of sending them in the clear generally outweigh any extra protection from keeping them secret. To save power, a sensor node may employ *early rejection* by turning off its radio after determining the message is not addressed to it. With broadcast messages, nodes can employ early rejection on the AM field as well. If the address and AM type are encrypted, early rejection cannot be invoked until after these fields are decrypted. This wastes power if rejection is frequent. Encrypting the length field adds little to security since the length of message can be inferred regardless.

To detect transmission errors, TinyOS senders compute a 16-bit cycle redundancy check (CRC) over the packet. The receiver recomputes the CRC during reception and verifies it with the received CRC field. If they are equal, the receiver accepts the packet and rejects it otherwise. However, CRCs provide no security against malicious modification or forgery of packets. To guarantee message integrity and authenticity, TinySec replaces the CRC with a MAC. The MAC protects the entire packet, including the destination address, AM type, length, source address and counter (if present), and the data (whether encrypted or not). This protects the data from tampering. It also prevents attackers from redirecting a packet intended for one node to another node, and prevents packet truncation and other attacks. Since MACs detect malicious changes, they also detect transmission errors, and TinySec does not require a CRC.

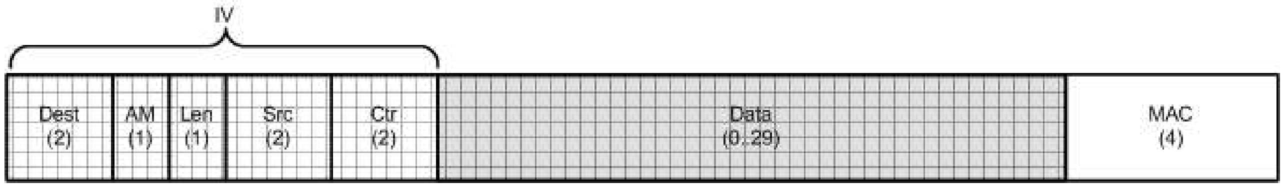
The TinyOS packet format contains a group field to prevent different sensor networks from interfering with each other. It can thought of as a kind of weak access control mechanism for non-malicious environments. Since TinySec enforces access control with a MAC, the group byte is unnecessary in TinySec. Instead, different networks should use different keys.

6. SECURITY ANALYSIS

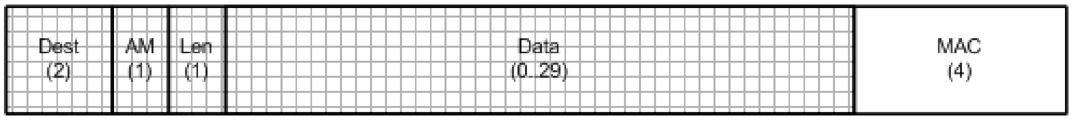
6.1 Message integrity and authenticity

The security of CBC-MAC is directly related to the length of the MAC. Conventional security protocols use MACs of 8 or 16 bytes, again erring on the side of caution. We show here that our choice of a 4 byte MAC is not detrimental in the context of sensor networks.

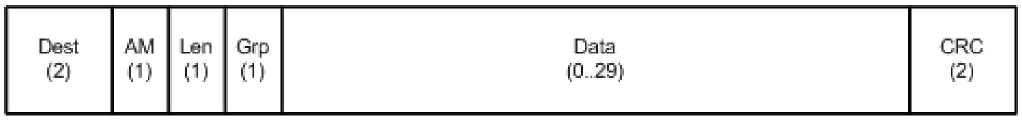
We can model TinySec's CBC-MAC as a function that produces 4 bytes of output [9]. Given a 4 byte MAC, then, an adversary has a 1 in 2^{32} chance in blindly forging a valid MAC for a particular message. If an adversary repeatedly attempts blind forgeries, she should succeed after about 2^{31} tries. Note that adversaries cannot determine off-line if a forgery will be successful or not; an adversary can only test the validity of an attempted forgery by sending it to an authorized receiver. This implies she must send about 2^{31} packets before she can succeed at forging the MAC for a single malicious packet. In conventional networks, this number



(a) TinySec-AE packet format



(b) TinySec-Auth packet format



(c) TinyOS packet format

Figure 1: The TinySec and TinyOS packet formats. The byte size of each field is indicated below the label. Fields that have been hatched are protected by the MAC. In TinySec-AE, the data field, shaded gray, is encrypted.

is not large enough for security. However, in sensor networks, this may provide an adequate level of security. Adversaries can try to flood the channel with forgeries, but on a 19.2kb/s channel, one can only send 40 forgery attempts per second, so sending 2^{31} packets at this rate would take over 20 months. Battery-operated sensor nodes do not have enough energy to receive that many messages. Furthermore, the adversary will have launched a quite effective denial of service attack since they need to occupy the radio channel for such a long time.

Clearly, it is both desirable and feasible to detect when such a attack is underway. A simple heuristic is probably sufficient: nodes could signal the base station when the rate of MAC failures exceeds some predetermined threshold.

6.2 Confidentiality

The security of CBC mode encryption reduces to the length of the IV, but this security assumes no IV reuse.[5] With an 8 byte counter or 16 byte random IV, avoiding repetition is relatively easy. Although TinySec uses an 8 byte IV, we limited ourselves to 4 additional bytes of overhead in the packet to represent the IV. The other 4 bytes of the IV borrow from the existing header fields: the destination address, the AM type, and the length.

TinySec partitions the last 4 bytes of the IV into *src*||*ctr*, where *src* is the source address of the sender and *ctr* is a 16 bit counter starting at 0. We selected this format because the other alternatives are insecure. Having every node generate IVs from a 4 byte counter starting at 0 is bad idea; the first packet sent from one node will reuse the same IV as the first packet sent from all other nodes. Generating IVs randomly is also a poor choice; the birthday paradox implies we can expect a collision after 2^{16} total packets in the network.

Our format for the last 4 bytes strives to maximize the number of packets each node can send before there is a global repetition of an IV value. The *src*||*ctr* format of the last 4 bytes guarantees each node can send at least 2^{16} packets before IV reuse occurs. For a network of n nodes all sending packets at approximately the same rate, this results in about $n \cdot 2^{16}$ total packets before we expect an instance of IV reuse. In conventional networks, end hosts transmitting at 1 Mb/s will send 2^{16} packets in less than an hour. However, since sensor nodes must conserve power to be long-lived, we envision the average data rate in most sensor networks will be dramatically less than conventional networks. For example, the sensors deployed at Great Duck Island send a reading once every 70 seconds [39]. We expect data rates to be on the order of 1 packet per minute, hour, or day. At one packet per minute per node, IV reuse will not occur for over 45 days.

IV reuse is only a problem when we reuse the same IV with the same key. If IV reuse is imminent, a key update protocol can be used to exchange new TinySec keys. Other researchers are currently exploring key update protocols in TinySec, and we do not address them here.

We specifically selected CBC mode for TinySec because of its robustness to information leakage when IVs repeat.

[5]The security of CBC mode also depends on the security of underlying block cipher, but we assume Skipjack is a secure block cipher.

In contrast to stream ciphers, where a repeated IV can reveal the plaintext of both messages, in CBC mode IV reuse reveals only the length (in blocks) of the longest shared prefix of the two messages. The first 4 bytes of the IV, $dst||AM||\ell$, help prevent information leakage during the unfortunate event of a counter on a node repeating. If a counter value for a particular source address is reused, there is only potential information leakage when the $dst||AM||\ell$ values are exactly the same for both messages. The means both messages were sent to the same destination and AM type, and both messages have the same length. Moreover, even in this case, information only leaks if both plaintexts agree in their first block. Consequently, information only leaks when one node sends two different packets with the same first 8 bytes and IV, to the same destination, with the same AM type, and of the same length.

In summary, the combination of carefully formatted IVs, low data rates, and CBC mode for encryption enables TinySec to provide strong confidentiality guarantees for applications in sensor networks. Conventional network security protocols are overly conservative because they can afford to be. In sensor networks we cannot afford this luxury. Fortunately, by selecting the right cryptographic primitives and using them carefully, we can tune down the security parameters and get the most out of the overhead.

7. KEYING MECHANISMS

A keying mechanism determines how cryptographic keys are distributed and shared throughout the network. The TinySec protocol is not limited to any particular keying mechanism; any can be used in conjunction with TinySec. In this section, we discuss the tradeoffs among different possible keying mechanisms in sensor networks. See Table 1 for a summary.

The appropriate keying mechanism for a particular network depends on several factors such as the target threat model, ease of use, and the networking and security requirements of applications. In cryptographic design, a good rule of thumb is to use different keys for different applications. When we refer to a *TinySec key*, we mean a pair of Skipjack keys, one for encrypting data, and one for computing MACs.

The simplest keying mechanism is to use a single network-wide TinySec key among the authorized nodes. A network-wide shared key provides a baseline level of security, maximizes usability, and minimizes configuration. Any authorized node can exchange messages with any other authorized node, and all communication is encrypted. Messages from unauthorized nodes are rejected. Key distribution is relatively simple; nodes are loaded with the shared key before deployment. This also makes it easy to secure legacy applications, since TinySec is transparent and can be enabled without disrupting existing code.

However, network-wide keying cannot protect against node capture attacks. If an adversary compromises a single node or learns the secret key, she can eavesdrop on traffic and inject messages anywhere in the network. To address the node capture threat, we need a keying mechanism with finer granularity.

A more robust option is for nodes to share a key for communication only if they need to communicate with each other. The simplest example of this idea is per-link keying, where we use a separate TinySec key for each pair of nodes who might wish to communicate. This provides better resilience against node capture attacks: a compromised node can only decrypt traffic addressed to it and can only inject traffic to its immediate neighbors. This approach has drawbacks. Key distribution becomes challenging, but recent research is beginning to address this issue [14, 16, 19, 28]. Also, per-link keying limits *passive participation* [26], a type of in-network processing where nodes take actions based on messages they overhear, and *local broadcast*, where nodes can cheaply send a packet to all their neighbors. Since a node cannot decrypt and authenticate messages not addressed to it, passive participation and local broadcast are incompatible with per-link keying.

A less restrictive approach is for groups of neighboring nodes to share a TinySec key rather than each pair. This enables passive participation and local broadcast, but key distribution and setup is still an issue. Group keying provides an intermediate level of resilience to node capture attacks: a compromised node can decrypt all messages from nodes in its group, but cannot violate the confidentiality of other groups' messages and cannot inject messages to other groups.

8. IMPLEMENTATION

We have implemented TinySec for the Berkeley sensor nodes. TinySec currently runs on the Mica, Mica2, and Mica2Dot platforms, each using Atmel processors; the Mica sensor node uses the RFM TR1000 radio, while the Mica2 and Mica2Dot nodes use the Chipcon CC1000 radio. Additionally, TinySec is integrated into the TOSSIM simulator, which runs on an Intel x86 platform. Others have ported TinySec to a Texas Instruments microprocessor. Given the broad range of platforms that TinySec runs on, we believe it will be easily portable to both new processors as well as new radio architectures.

We implemented TinySec in 3000 lines of nesC code [21], the programming language used for TinyOS. Our implementation of TinySec requires 728 bytes of RAM and 7146 bytes of program space.[6]

We modified the default TinyOS 1.1.2 radio stack to incorporate TinySec. We modified the stack to re-direct byte level radio events to the `TinySecM` module.

Integration into TinyOS required some modifications to the scheduler. When the media access control layer successfully acquires the channel, it signals `TinySecM`. At this point, the security module begins the cryptographic computations. The cryptographic computations must be completed by the time the radio finishes sending the start symbol. To achieve the real-time deadline, we modified the TinyOS scheduling process. TinyOS provides a rudimentary form of process management. *Tasks* run until completion in FIFO order. One option for implementing TinySec would be to submit a task with the cryptographic operations to the scheduler. However, if the task queue is non-empty, the cryptographic operations may not complete in time, since they must wait until the task queue empties. We instead implement a two-priority scheduler, where cryptographic operations are run

[6]We have subsequently improved the RAM usage of our TinySec implementation, which requires 256 bytes of RAM and 8152 bytes of ROM. Our optimizations save 472 bytes of RAM at the expense of 6% slower block cipher operations. All performance results in this paper use the old implementation, but we do not expect significant differences with our new implementation.

Keying mechanism	Benefits	Costs
Single network-wide key	Simple; easy to deploy; supports passive participation and local broadcast	Not robust to node compromise
Per-link keys between neighboring nodes	Graceful degradation in the presence of compromised nodes	Needs a key distribution protocol; prohibits passive participation and local broadcast
Group keys	Graceful degradation in the presence of compromised nodes; supports passive participation and local broadcast	Requires key distribution; trades off robustness to node compromise for added functionality

Table 1: A summary of different keying mechanisms for link-layer security.

with high priority and all other tasks run at low priority. The two-priority scheduler ensures that cryptographic operations complete on time so that encryption and decryption execute concurrently with packet transmission and reception.

TinySec is cipher independent. We have implemented both RC5 and Skipjack and can switch between them without difficulty.

Since the maximum data payload in TinyOS is 29 bytes, we can safely use the upper two bits of the length byte to indicate which level of protection is enabled on that packet: TinySec-AE, TinySec-Auth, or unprotected TinyOS packets. When sending a packet, the TinySec stack encodes the TinySec mode of the packet into the length bit.

For ease of deployment, we implemented a network-wide shared key model. We modified the application build process to include the key at compile time to ease key deployment hassles. The build process maintains a key file at the developer's machine and uses a key from the file. Other researchers have extended TinySec to use finer grained keying mechanisms. Refer to Section 9.2 for further details.

To enable the TinySec stack, an application writer needs only to specify "`TINYSEC=true`" on the command line to `make` (or in the `Makefile`). Messages sent in this configuration will then be sent using TinySec-Auth. We chose this as the default TinySec mode since it imposes a minimal amount of overhead (Section 9.1). An application writer can send authenticated and encrypted packets with TinySec-AE by simply making a function call to switch modes.

TinySec is currently distributed with the official TinyOS releases.

9. EVALUATION

9.1 Measurements

TinySec increases the computational and energy costs of sending a packet. For application writers to adopt TinySec, these costs must be modest compared to the benefits that TinySec provides. There are two main components to these costs: larger packet sizes when using TinySec, and the extra computation time and energy needed for cryptography. The costs due to increased packet length will be borne by all implementations, even those using cryptographic hardware. Naturally, the computation costs will vary based on the implementation.

To analytically estimate the costs of the cryptography, we first calculate the effect of packet lengths in TinySec. Recall that TinySec increases packet lengths by 1 or 5 bytes (according to whether TinySec-Auth or TinySec-AE is in use). Longer packets cost us in several ways: first, they reduce bandwidth; second, because the communications channel is fairly slow, they increase latency; third, they increase energy consumption, because the radio must be turned on longer when transmitting longer packets. We first calculate the expected contribution to TinySec's overhead that comes solely from increased packet sizes. Table 2 shows the extra time needed to transmit a packet using TinySec. We expect TinySec-AE to increase packet latencies (compared to the current TinyOS stack) by 8.0%; for TinySec-Auth, the corresponding figure is 1.5%. Note that sending a packet involves sending many more bits than just the data and its associated header; as a part of the media access control protocol, a 28 byte start symbol and additional synchronization bytes are also sent. This reduces the impact of adding an additional byte of overhead to a header, since there is a high fixed cost for sending a packet.

Cipher & Impl.	Time (ms)	Time (byte times)
RC5 (C)	0.90	2.2
Skipjack (C)	0.38	0.9
RC5 (C, assembly)	0.26	0.6

Table 3: Time to execute cipher operations on the Mica2 sensor nodes. We display the time both in milliseconds and in byte times.

Next, we implemented TinySec and experimentally measured its performance costs. In these experiments, we empirically determined TinySec's impact on bandwidth, energy, and latency on the Berkeley Mica2 sensor nodes. We used a variety of microbenchmarks and macrobenchmarks to evaluate TinySec. In addition to allowing us to obtain latency figures, the macrobenchmark allowed us to evaluate the difficulty in enabling TinySec for an existing, large sensor network application.

We introduce the term *byte time* to refer to the duration that it takes to transmit a single byte of data over the radio. On the Mica2, a byte time is 0.42 ms. Measuring time in byte times is a convenience that allows us to relate the time of an operation to the packet length.

Cipher performance. We tested the performance of two 64-bit block ciphers, Skipjack and RC5, to determine their speed. We must be able to complete a block cipher operation quickly since the cryptographic operations are overlapped with the radio operations; if the cipher operation doesn't complete in time, the data needed for the radio will not be available. More importantly, faster block ciphers consume less energy.

	Application Data (b)	Packet Overhead (b)	Total Size (b)	Time to Transmit(ms)	Increase Over Current TinyOS Stack
Current TinyOS Stack	24	39	63	26.2	—
TinySec-Auth	24	40	64	26.7	1.6%
TinySec-AE	24	44	68	28.3	7.9%

Table 2: Table listing the expected latency overhead incurred by TinySec. The packet overhead includes space needed for the header and media access control information, such as the start symbol. Since TinySec increases the packet size by a fixed amount, it will increase the time needed to send the packet over the radio. This impacts bandwidth, latency, and the energy needed to send a packet. We confirm this predicted overhead increase experimentally.

	Energy (mAH)	Increase
Current TinyOS Stack	0.000160	—
TinySec-Auth	0.000165	3%
TinySec-AE	0.000176	10%

Table 4: Total energy consumed to send a 24 byte packet.

The results in Table 3 indicate that both RC5 and Skipjack are reasonable choices for use in link layer security: each block cipher operation takes less than a byte time. Because each such operation operates on 8 bytes, this is reasonably fast. We use the rule of thumb that the block cipher operation should complete in under a few byte times. If it does not, we can encounter problems when the radio must wait for the processor to complete the cryptographic operation. Note that our initial C-only version of RC5 was not fast enough in all cases, so we optimized its inner loop using in-line assembly code to get better performance. We have not optimized Skipjack similarly, but we believe it would be possible to improve Skipjack's performance so that it is fairly competitive with RC5. As we mentioned, we settled on Skipjack for the default TinySec configuration even though it is slower than our best RC5 implementation since it has minimal key setup costs and is free from patent issues. We use Skipjack as our cipher for all of our experiments.

Energy costs. To determine the energy overhead in using TinySec, we sampled the instantaneous current drawn by a transmitter sending 24 bytes of application data. Figure 2 shows the current as a function of time for sending the packet. We provide graphs for the current TinyOS stack (no security), TinySec-Auth, and TinySec-AE.

The radio exposes a byte-level interface. Small periodic spikes occur once every byte time. When the radio is in transmit mode, the radio stack delivers a new byte to the radio once every byte time. When the radio isn't transmitting, the stack samples the radio once a byte time looking for the start symbol from a new packet.

There are a number of features of the graph that illustrate TinySec's implementation. The large power draw at the start of sending a packet for both TinySec graphs is due to the cryptographic operations. When sending a packet, TinySec overlaps the MAC and encryption computation with the sending of the start symbol. The start symbol lasts for 28 bytes. For the largest possible packet size using TinySec-AE, a maximum of 10 block cipher operations are required,

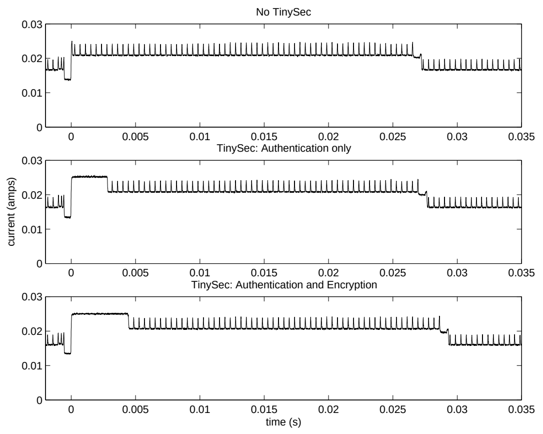


Figure 2: The power consumption for sending a packet. All packets contained 24 byte payloads. The top graph shows the power consumption when sending the packet with the current TinyOS stack (no security). In the middle graph, we use TinySec-Auth, while the bottom graph uses TinySec-AE. Notice the large power draw at the beginning of sending as the encryption and MAC computation is overlapped with the sending of the start symbol. Additionally, note that when sending with TinySec, the packets are larger in length.

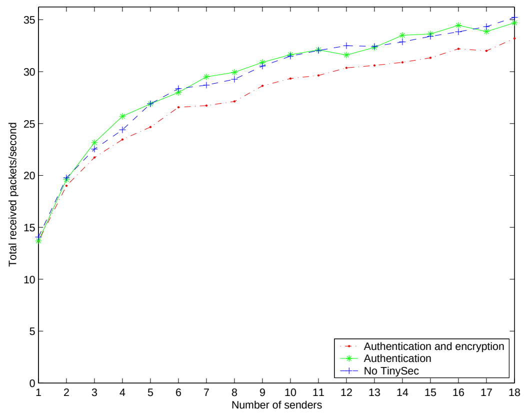


Figure 3: Bandwidth, plotted as a function of the number of send-receive pairs. We compare TinySec-Auth and TinySec-AE to the bandwidth without using TinySec.

hence the block cipher must take no more than $28/10 = 2.8$ byte times per operation. The block cipher uses the processor heavily, leading to a large initial power draw while the packet is encrypted. As one would expect, the computation period is larger when using encryption since there are more block cipher operations.

The power consumption measurements show that the sending period for TinySec-AE is about 5 byte times longer than without TinySec. Also, TinySec-Auth is longer by a single additional byte time. This is due to the extra packet overhead that TinySec imposes.

After integrating to calculate the area under the curves for the sending period, we find that the total energy consumed for sending a packet with the default stack is 0.00016 mAH. Using TinySec-Auth, it is 0.000165 mAH, a 3% increase. Using TinySec-AE, the total energy to send a packet is 0.000176 mAH, a 10% increase over the energy costs for sending a packet without TinySec.

Comparing these figures to Table 2 illustrates an interesting point: the cryptographic operations do not consume a significant amount of energy. TinySec's energy costs come from two sources: increased packet lengths, and extra computation from the cryptography. Using the fact that TinySec-AE performs about twice as many block cipher operations as TinySec-Auth, it is possible to estimate the relative magnitude of these two sources of energy overhead. We estimate that, of TinySec-Auth's measured 3% energy overhead, roughly 1% comes from increased packets lengths and 2% from extra computation. Of TinySec-AE's measured 10% energy overhead, perhaps 6% comes from increased packet lengths and 4% from extra computation. Our measurements provide an upper bound on the energy savings that hardware support for cryptography could provide: at best, hardware could eliminate the energy costs due to the cryptographic computations, but even hardware-accelerated versions of TinySec would still have to pay the energy overhead associated with TinySec's increased packet lengths.

This demonstrates that hardware support is not a precondition for efficient link-layer cryptography in sensor networks; software cryptography achieves acceptable energy costs. Also, as we have shown, even hardware-assisted cryptography would not perform significantly better than TinySec's software-only implementation.

Throughput. To measure the maximum throughput when using TinySec, we computed the total number of packets that could be sent in a 30 second time period. In this experiment, we configured a network of nodes so that multiple nodes would simultaneously transmit as rapidly as possible. Since the number of senders affects the channel utilization, we varied the number of senders. This allows us to characterize the throughput at different regimes. We sent 24 bytes of application data using the current TinyOS stack, TinySec-Auth, and TinySec-AE. We measured the number of packets that were successfully received. The results are in Figure 3.

TinySec-Auth's bandwidth characteristics are nearly identical to those of the current TinyOS stack. TinySec-AE, with its 5 byte larger packets, consistently achieved 6% lower throughput when more than 5 senders participated. With fewer senders, channel contention is less of an issue, so the packet length overhead does not affect the throughput. We expect sensor networks to largely operate in the latter regime,

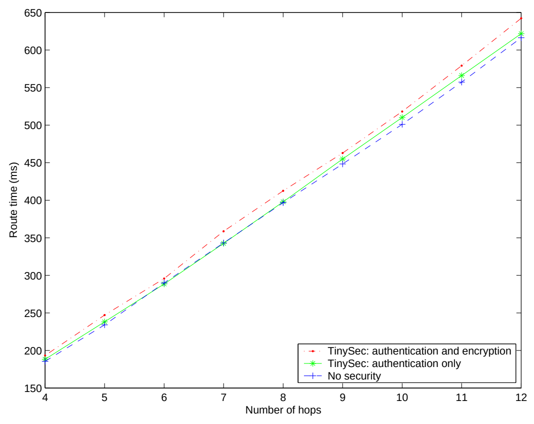


Figure 4: End-to-end latency in a large system using TinySec. We measured the time to route a message over a number of hops. We used 37 nodes in this experiment. Note that TinySec increases end-to-end latency. As shown in Figure 5, the extra latency can be fully explained by TinySec's impact on packet sizes: longer packets take longer to transmit and hence increase latency.

since at higher bandwidths, the lifetime of the network will be limited.

As further confirmation that the throughput difference is only due to the differences in packet length and not the computational costs, we reran the experiments with the total packet size adjusted to be the same for all three radio stacks. We compared TinySec-AE with a 24 byte data payload, TinySec-Auth with a 28 byte data payload, and the current TinyOS stack with a 29 byte payload so that all three packets were 36 bytes in total. We then reran the bandwidth tests, again varying the number of senders, and all three stacks delivered the same bandwidth. We thus conclude that the bandwidth difference is fully explained by the difference in packet sizes, and not due to any increased computation costs.

Latency macrobenchmarks. For our final test, we integrated TinySec into a large existing TinyOS sensor network application. The test had two purposes: we wanted to measure the latency when routing packets and to see how usable TinySec is with a large existing system.

The NEST Pursuit-Evasion Midterm Demo Game is a sensor network application that tracks the movements of a mobile evader using magnetometers. The sensor nodes are deployed in a field and form a landmark based routing tree [40]. Separately, autonomous mobile pursuer robots collect data from the sensor networks to follow the evaders. Nodes that have magnetometer readings from the the evader route the data to a central *landmark* node that routes it to the mobile pursuers.

In our test, we set up 36 nodes in a 6 foot by 7.75 foot grid. The total network diameter was 10 hops. Additionally, we used a base station connected to a computer to control the network and snoop on traffic. For our tests, we measured the time it took for a message to go from the base station to a sensor network node and get routed through the landmark to another node which broadcasts a reply back to the base

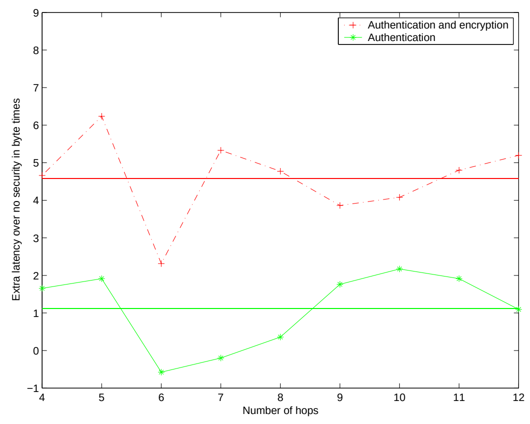


Figure 5: The increase in latency when routing packets using TinySec. We display the results in *byte times*, the time it takes to transmit one byte of information over the radio. This shows that TinySec's impact on end-to-end latency is caused by the increased length of TinySec packets. There is a close correspondence between theory and practice: using authentication and encryption increases packet length by 5 bytes, and empirically, we see that latency is increased by 4.6 byte times; similarly, using authentication alone increases packet lengths by 1 byte, and empirically it increases latency by 1.1 byte times. Note that we have normalized by the route distance.

station. Since all routes in our landmark routing scheme pass through the landmark, sending a message from A to B takes at least two hops. Additionally, the base station must send a message to A telling it to initiate its transmission, and B sends a message back to the base station. This means that all paths in our test are at least 4 hops in length.

We measured the round-trip time it took to transmit packets across paths of different hop lengths. For each route length, we routed 200 packets with the current TinyOS radio stack (no security), with TinySec-Auth, and with TinySec-AE. The media access control protocol waits a uniformly random amount of time before sending the packet. The sum of many uniformly distributed random variables approaches a normal distribution, which is what we see with the longest routes. Note that the carrier sense in the media access control protocol is occasionally wrong, so that it backs off for a longer time; this leads to a heavy tail in the distribution.

To compare the different radio stacks to each other, we average the individual experimental results to obtain an average single route time per route length. We plot the results in Figure 4. Each data point represents the average elapsed time needed to transmit the packets across a route of a given hopcount. Routing with TinySec-Auth takes longer than with the current TinyOS radio stack; routing with TinySec-AE takes longer than both. As one example, across 12-hop routes, the total difference between TinySec-AE and the current TinyOS stack is 26 ms. As before, the latency difference can be explained by the increased packet size.

We would expect the total time to transmit a packet to increase by 5 byte times per link when using TinySec-AE. Since each byte time is 0.42 ms, over a 12 hop route, latency is expected to be 25.20 ms worse; we observed a 26 ms difference. We quantify this effect more precisely in Figure 5. We compare the average byte time difference between the two TinySec modes and the current TinyOS radio stack. We normalize the result by the hop count. The outliers from the heavy tail skew the average, so we remove those values from consideration. Recall the heavy tail is due to the media access control protocol backing off erroneously, and are not material to the metric we are measuring. In the chart, then, we see byte time differences that we measure match our expectations: it takes an extra 5 byte times to transmit a TinySec-AE packet one hop, and 1 extra byte-time to transmit a TinySec-Auth packet.

Summary. In all cases, the energy, bandwidth, and latency overhead of using TinySec is less than 10%. Much of the overhead can be fully explained by the increased packet length that TinySec imposes. There is an additional energy cost to performing the cryptographic computations, amounting to less than 1/2 of the total energy increase of TinySec-AE. Thus, TinySec will be very competitive with hardware solutions.

9.2 Ease of Use

We use an indirect means to evaluate TinySec's ease of use. In the first test, noted above in our latency measurements, we integrated TinySec into a large, existing application. Including TinySec did not require any changes to the application code and only required a one line change to the makefile. Other applications can enable TinySec with the same ease.

TinySec has gathered a number of external users. In particular, we are aware of five other projects that are using TinySec as a research platform to enable their research in key distribution. TinyPK uses RSA to authenticate network downloaded code and to exchange TinySec keys [43]; it is built on top of TinySec, and relies upon TinySec for link-layer security. TinyCrypt, still in development at Harvard, aims to use elliptic curve cryptography to exchange TinySec keys for the Mica2 sensor nodes [32]. Meanwhile, researchers at SRI have designed a scheme for key exchange, group management, and key revocation [18]; they use TinySec's packet format for transport-level security. Also, SecureSense provides dynamic security service composition using the TinySec infrastructure [46], and the Bosch corporation used a modified version of TinySec to implement a prototype burglar alarm security system. Finally, our TinySec implementation is included in the current TinyOS public release and is available for routine use as well.

10. RELATED WORK

GSM, IEEE 802.11, and Bluetooth. The GSM frame format was intended to provide confidentiality (but not integrity) protection of voice data with little overhead. Unfortunately, researchers have found serious vulnerabilities with the GSM security mechanisms [7]. The 802.11 wireless networking standard initially specified WEP, a scheme that used RC4 encryption for confidentiality and a CRC checksum for integrity protection. However, security researchers quickly found WEP to be thoroughly flawed [12, 20, 38, 42]: its 24-bit IVs are too short; the CRC checksum fails to protect integrity; and, naive use of the IV to diversify RC4's key enables devastating cryptanalytic related-key attacks. Subsequently, the standards group has designed TKIP, an interim replacement for WEP with stronger message authentication and more careful mixing of the IV with the key. However, TKIP is only designed as a short-term band-aid, and its per-packet overhead is substantial. The long-term successor to WEP and TKIP will be 802.11i's CCMP, which uses AES in CCM mode, 48-bit IVs, and a strong 64-bit message authentication code. CCMP appears to be well-designed, but unfortunately for our purposes, the per-packet overhead is too high to be practical for use in sensor networks [13]. The Bluetooth specification also includes a cryptographic security mechanism, but this has also been shown to be flawed [25].

SNEP. The closest previous work is SNEP [33], which specifically targets sensor networks, but SNEP was unfortunately neither fully specified nor fully implemented. Also, each recipient must maintain a counter for each sender communicating with it. Managing this state encounters similar complexity as maintaining replay counters, as we discussed in Section 3.1.

IEEE 802.15.4. Recently, the IEEE adopted the 802.15.4 standard, specifying a physical and media access control layer for low data rate wireless applications [6]. Vendors are starting to sell sensor nodes equipped with this radio platform [1, 2].

The 802.15.4 standard includes provisions for link-layer security. Many features of their architecture are similar to TinySec, although there are important differences. For example, 802.15.4 specifies a stream cipher mode of encryption, and to avoid IV reuse they require a larger IV. They have also chosen to include replay protection, as an optional feature, into the link-layer security package.

IEEE 802.15.4 radio chips perform all of their computations in hardware, reducing energy consumption and CPU utilization. However, as we have shown, in retrospect the use of hardware was not strictly necessary for security: software cryptography could have been used with only a small increase in energy consumption.

Sastry and Wagner have found a few problems with some of the optional modes in the 802.15.4 specification and the feasibility of supporting different keying models [34]. Despite the presence of these defects, the 802.15.4 security architecture is sound. It includes many well designed security features and presents a step forward for embedded device wireless security. Proper use of the security API can lead to secure applications.

11. CONCLUSION

TinySec addresses security in devices where energy and computation power present significant resource limitations. We have designed TinySec to address these deficiencies using the lessons we have learned from other security protocols. We have tried to highlight our design process from a cryptographic perspective that meets both the intended resource constraints and security requirements. TinySec relies on cryptographic primitives that have been vetted in the security community for many years.

Our TinySec implementation is in wide use throughout the sensor network community. We know of researchers building key exchange protocols on top of TinySec. Others have ported TinySec to their own custom hardware. TinySec is simple enough to integrate into existing applications that the burden on application programmers is minimal.

Finally, we have extensively measured the performance characteristics of TinySec. Its energy consumption, even when used in the most resource-intensive and most secure mode, is a modest 10%. Using TinySec-Auth, the extra energy consumed is a scant 3%. Similarly low impacts on bandwidth and latency prove that software based link layer security is a feasible reality for devices with extreme resource limitations.

Acknowledgements

Robert Szewczyk helped us measure the power consumption of TinySec. This work was supported in part by research grants and contracts from the Defense Advanced Research Project Agency (NEST contract F33615-01-C-1895), the National Science Foundation, and the US Postal Service, and gifts from Bosch and Intel.

12. REFERENCES

[1] Crossbow technology inc. `http://www.xbow.com`.

[2] Moteiv. `http://www.moteiv.com/`.

[3] OpenSSL. `http://www.openssl.org`.

[4] Security architecture for the Internet Protocol. RFC 2401, November 1998.

[5] Smart buildings admit their faults. Lab Notes: Research from the College of Engineering, UC Berkeley, `http://www.coe.berkeley.edu/labnotes/1101smartbuildings.html`, November 2001.

[6] Wireless medium access control and physical layer specifications for low-rate wireless personal area networks. IEEE Standard, 802.15.4-2003, May 2003. ISBN 0-7381-3677-5.

[7] Elad Barkan, Eli Biham, and Nathan Keller. Instant ciphertext-only cryptanalysis of GSM encrypted communication. In *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, 2003.

[8] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption: Analysis of the DES modes of operation. In *Proceedings of 38th Annual Symposium on Foundations of Computer Science (FOCS 97)*, 1997.

[9] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of the cipher block chaining message authentication code. *Journal of Computer and System Sciences*, 61(3):362–399, December 2000.

[10] Steven M. Bellovin. Problem areas for the IP security protocols. In *Proceedings of the Sixth USENIX Security Symposium*, 1996.

[11] Steven M. Bellovin and Matt Blaze. Cryptographic modes of operation for the internet. In *Second NIST Workshop on Modes of Operation*, August 2001.

[12] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting mobile communications: The insecurity of 802.11. In *The Seventh Annual International Conference on Mobile Computing and Networking (MobiCom 2001)*, 2001.

[13] Nancy Cam-Winget, Russ Housley, David Wagner, and Jesse Walker. Security flaws in 802.11 data link protocols. *Communications of the ACM*, 46(5):35–39, May 2003. Special Issue on Wireless Security.

[14] Haowen Chan, Adrian Perrig, and Dawn Song. Random key predistribution schemes for sensor networks. In *IEEE Symposium on Security and Privacy*, May 2003.

[15] E. Dawson and L. Nielsen. Automated cryptanalysis of XOR plaintext strings. *Cryptologia*, (2):165–181, April 1996.

[16] Wenliang Du, Jing Deng, Yunghsiang S. Han, and Pramod K. Varshney. A pairwise key pre-distribution scheme for wireless sensor networks. In *10th ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[17] G.L. Duckworth, D.C. Gilbert, and J.E. Barger. Acoustic counter-sniper system. In *SPIE International Symposium on Enabling Technologies for Law Enforcement and Security*, 1996.

[18] Bruno Dutertre, Steven Cheung, and Joshua Levy. Lightweight key management in wireless sensor networks by leveraging initial trust. Technical Report SRI-SDL-04-02, SRI International, April 2004.

[19] Laurent Eschenauer and Virgil D. Gligor. A key-management scheme for distributed sensor networks. In *9th ACM Conference on Computer and Communication Security (CCS)*, November 2002.

[20] Scott Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of RC4. *Lecture Notes in Computer Science*, 2259:1–24, 2001.

[21] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to network embedded systems. In *Programming Language Design and Implementation (PLDI)*, June 2003.

[22] Mohamed G. Gouda, E.N. Elnozahy, Chin-Tser Huang, and Tommy M. McGuire. Hop integrity in computer networks. *IEEE/ACM Transactions on Networking*, 10(3):308–319, June 2002.

[23] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of ACM ASPLOS IX*, pages 93–104, November 2000.

[24] Chris Hurley. The worldwide wardrive: The myths, the misconceptions, the truth, the future. In *Defcon 11*, August 2003.

[25] Markus Jakobsson and Susanne Wetzel. Security weaknesses in Bluetooth. In *CT-RSA 2001*, pages 176–191. Springer-Verlag, 2001. LNCS 2020.

[26] Chris Karlof, Yaping Li, and Joe Polastre. ARRIVE: Algorithm for robust routing in volatile environments. Technical Report UCB/CSD-03-1233, University of California at Berkeley, May 2002.

[27] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*. Springer-Verlag Heidelberg, January 2001.

[28] Donggang Liu and Peng Ning. Establishing pairwise keys in distributed sensor networks. In *10th ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[29] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. In *The Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, 2002.

[30] Samuel R. Madden, Robert Szewczyk, Michael J. Franklin, and David Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Workshop on Mobile Computing and Systems Applications*, 2002.

[31] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, and David Culler. Wireless sensor networks for habitat monitoring. In *First ACM International Workshop on Wireless Sensor Networks and Applications*, 2002.

[32] David Malan, Matt Welsh, and Michael D. Smith. A public-key infrastructure for key distribution in TinyOS based on elliptic curve cryptography. In *First IEEE International Conference on Sensor and Ad Hoc Communications and Networks*, October 2004.

[33] Adrian Perrig, Robert Szewczyk, Victor Wen, David Culler, and J.D. Tygar. SPINS: Security protocols for sensor networks. In *The Seventh Annual International Conference on Mobile Computing and Networking (MobiCom 2001)*, 2001.

[34] Naveen Sastry and David Wagner. Security considerations for IEEE 802.15.4 networks. In *ACM Workshop on Wireless Security (WiSe 2004)*, September 2004.

[35] Bruce Schneier. *Applied Cryptography, Second Edition*. John Wiley & Sons, 1996.

[36] Peter Shipley. Open WLANs: the early results of wardriving, 2001.

[37] Peter Shipley, 2003. personal communication.

[38] Adam Stubblefield, John Ioannidis, and Aviel D. Rubin. Using the fluhrer, mantin, and shamir attack to break WEP. In *Network and Distributed Systems Security Symposium (NDSS)*, 2002.

[39] Robert Szewczyk, Joseph Polastre, Alan Mainwaring, and David Culler. Lessons from a sensor network expedition. In *First European Workshop on Wireless Sensor Networks (EWSN '04)*, January 2004.

[40] P.F. Tsuchiya. The landmark hierarchy: a new hierarchy for routing in very large networks. *ACM Computer Communication Review*, 18(4):35–42, 1988.

[41] Ramnath Venugopalan, Prasanth Ganesan, Pushkin Peddabachagari, Alexander Dean, Frank Mueller, and Mihail Sichitiu. Encryption overhead in embedded systems and sensor network nodes: Modeling and analysis. In *2003 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 188–197, 2003.

[42] Jessie Walker. Unsafe at any key size; an analysis of the WEP encapsulation. `http://grouper.ieee.org/groups/802/11/Documents/DocumentHolder/0-362.zi%p`.

[43] Ron Watro, Derrick Kong, Sue fen Cuti, Jen Mulligan, Charlie Gardiner, and Dan Coffin. TinyPK. `http://www.is.bbn.com/projects/lws-nest/`.

[44] Matt Welsh, Dan Myung, Mark Gaynor, and Steve Moulton. Resuscitation monitoring with a wireless sensor network. Supplement to Circulation: Journal of the American Heart Association, October 2003.

[45] WiGLE. Wireless geographic logging engine—general stats, December 2003.

[46] Qi Xue and Aura Ganz. Runtime security composition for sensor networks (SecureSense). In *IEEE Vehicular Technology Conference (VTC Fall 2003)*, October 2003.

[47] T. Ylonen. SSH - secure login connections over the Internet. In *Proceedings of the Sixth USENIX Security Symposium*, 1996.