Data and Viral Code Integrity in Maté

Philip Levis
Computer Science Division
EECS Department
University of California
Berkeley, California
pal@cs.berkeley.edu

ABSTRACT

The Maté virtual machine is a powerful mechanism for sensor networks, providing execution safety, energy-efficiency through concise programs, and rapid network reprogramming through viral code. Viral code propagation has obvious security issues; a single nefarious program can quickly take over a network, while program conciseness exposes possible weaknesses to cryptanalysis and mote failure means that physical mote compromises can easily go undetected. Data replay attacks can disrupt the results from a network.

We present the security mechanisms we have incorporated into Maté to address these problems. We discuss the mechanisms with and without the possibility of mote compromise. We show how the disparate encrypt/decrypt difficulty of RSA can be used for public-key digital signatures, how cut-and-paste attacks can be prevented through version vectors, and data replay can be suppressed. These mechanisms suggest what roles symmetric and asymmetric cryptography can play in a sensor network, indicating what hardware primitives are needed for a sensor network to be secure.

1. INTRODUCTION

Wireless sensor networks pose novel problems in system management and programming. Networks of hundreds or thousands of nodes, limited in energy and individual resources, must be reprogrammable in response to changing needs. Limited energy budgets require concise programs, which can be written for a high-level virtual machine. Autonomous viral code propagation can make installing programs simple and rapid. Networks may require security in the presence of a slew of attacks, including snooping, mote compromise, partitioning, failure, traffic injection, and traffic control. Decentralized programming requires distributed and in-network security mechanisms as opposed to centralized ones.

In this work, we consider Maté, a tiny virtual machine designed for TinyOS sensor networks [6, 13]. High level Maté bytecodes allow complete sensing programs to be written in a few bytes; sending a packet is a single instruction. Maté programs are broken up into small capsules of instructions that virally propagate through a network using version numbers. Introducing a single mote running a new Maté program will cause the program to self-replicate through a network until the entire network is running the new program.

This viral code propagation makes reprogramming very simple, but raises clear security issues: a single nefarious code capsule can take over a network. Additionally, Maté can run capsules in response to data packets; an adversary can, in essence, execute Maté code by sending data. A spectrum of security requirements emerges, ranging from open operation to program integrity to complete code and data integrity as well as confidentiality.

Current TinyOS sensor networks use the `mica` platform, which has a 4MHz 8-bit microcontroller as a CPU, 4 KB of RAM, 128 KB of program memory, a 40 Kbit radio and 512 KB of non-volatile storage (EEPROM). The microcontroller has several security features. A fuse bit can be set that disables reading program memory from the chip; clearing the fuse bit erases the program memory. Also, the EEPROM is cleared when a new program is uploaded into program memory. These two features mean that an adversary cannot read data off the chip with normal software tools [1], making symmetric key cryptography very appealing. This is the basis of TinySec, a secure data-link layer implemented in TinyOS [8]. A network-shared secret key can be included in the network's program; even taking physical control of a mote won't allow an adversary to read the key using software.

However, a determined adversary could use hardware tools (examining the chip itself) or instrumentation (power analysis) to gain information about a running program [11, 12, 14]; additionally, there is no assurance that these hardware security mechanisms will be present in future mote designs. Therefore, just as with security requirements, one must consider a spectrum of adversary determination.

We present the security mechanisms we have introduced to Maté to address with these spectra of adversary determination and security requirements. Some (such as public-key cryptography) are currently unfeasible to fully implement in software, but we have implemented limited versions (e.g. small keys). These limitations indicate directions for future mote architectures if security is to be a high-priority requirement.

This work has several contributions. First, we define a hierarchy of security requirements for Maté networks. Second, we categorize the classes of attacks that could be launched against a Maté network. Third, we identify characteristics of the VM that could be weaknesses to cryptanalytic attacks. Fourth, we present mechanisms that meet the security requirements in the presence of the identified attacks and weaknesses.

Section 2 provides a broad overview of sensor networks and the aspects of Maté that are pertinent to discussing its security. In Section 3, we discuss our security analysis of Maté, categorizing VM security requirements, attacks, and weaknesses. Section 4 presents the security mechanisms we have introduced to Maté.

2. SENSOR NETWORKS

The progression of Moore's Law has led to the design and manufacture of small sensing computers that can communicate with one another over a wireless network [6]. Research and industry indicate that motes will be used in networks of hundreds, thousands, or more [7][5][22].

Sensor networks are distinct from traditional computing domains. Their design assumes being embedded in common environments (e.g., a corn field, a bathroom), instead of dedicated ones (e.g., a server room, an office). Mean time to failure combined with large numbers leads to routine failure; a network must be easy to repopulate without interrupting operation. Unlike infrastructure systems in controlled server room, the embedded nature of motes makes them physically vulnerable; an adversary can just pick one up and walk away with it for later analysis.

Energy is the most critical resource in a sensor network. One can easily recharge a laptop or a handheld; recharging thousands of motes (or even *finding* all of them!) is much more difficult. Because of its power requirements, communication in sensor networks is more precious than in other computing domains; sending a single bit of data can consume the energy of executing a thousand instructions.

2.1 TinyOS

TinyOS is an operating system designed specifically for use in sensor networks [6]. Combined with a family of wireless sensor devices, TinyOS is currently used as a research platform by over 70 groups worldwide. TinyOS has a simple event-based concurrency model. This model TinyOS allows high parallelism with low overhead, in contrast to a thread-based concurrency model in which thread stacks consume precious memory while blocking on a contended service.

The top-level TinyOS packet abstraction is an Active Message [21]. The characteristics of this abstraction are important because they define the capabilities of systems built on top of it. AM packets are an unreliable data link protocol; the TinyOS networking stack handles media access control and single hop communication between motes. Higher layer protocols (e.g. network or transport) are built on top of the AM interface.

AM packets can be sent to a specific mote (addressed with a 16 bit ID) or to a broadcast address (`0xffff`). TinyOS provides a namespace for up to 256 types of Active Messages, each of which can each be associated with a different software handler. AM types allow multiple network or data protocols to operate concurrently without conflict.

TinySec is an Active Messages implementation designed for security; it provides cryptographic packet confidentiality and integrity. TinySec uses the RC4 block cipher over the data region of the packet, with a 16-bit initialization vector sent in cleartext to provide confidentiality for repeated messages. The destination address, AM type, and packet length field as also sent in cleartext. The RC4 symmetric key is installed on each mote before deploying the network and resides in program memory. Integrity is provided by a full packet MAC, which uses a different key than data encryption.

2.2 Maté

Maté is a bytecode interpreter that runs on TinyOS. Code is broken in **capsules** of 24 instructions, each of which is a single byte long; larger programs can be composed of multiple capsules. In addition to bytecodes, capsules contain identifying and version information. There are three execution contexts that can run concurrently at instruction granularity. Maté capsules can forward themselves through a network; version numbers are used to determine if a heard capsule should be installed. Maté provides both a built-in ad-hoc routing algorithm (the `send` instruction) as well as mechanisms for writing new ones (the `sendr` instruction).

Initial versions of Maté had an explicit capsule forwarding instruction `forw`; experimental results showed this to be problematic. Current versions of Maté use a combination push/pull system inspired by global index work on PlanetP [4].

Each mote periodically sends out a version vector packet, containing the version numbers of the capsules installed. The send rate decays linearly over time; the period increases by one second on each send. If a mote hears a vector with an older version number than what it has, it starts transmitting the newer version of the capsule, sending it three times. If a mote hears a vector with a newer version number than what it has, it resets its version vector send period; the intention is that the mote with the newer capsule will hear the vector and start sending the code. Maté resets its version vector send period whenever it installs a new capsule.

Our initial experiments show this code forwarding mechanism to be much more efficient and effective than explicit forwarding; a stable network can enter a quiescent state in which version vectors are only send every few minutes. However, as soon as a new capsule enters the system, resetting the vector send period causes it to very quickly propagate through the entire network. Similarly, because Maté never stops sending version vectors, a disconnected mote that reconnects will be able to learn about and request code updates.

3. MATÉ SECURITY ANALYSIS

Maté's viral code propagation makes installing a new program simple and rapid; this functionality, however, can be used by an adversary to take control of a network just as easily and quickly. This network-centric programming poses several specific security threats to a network. Maté has very specific traffic and data patterns that can be used by an adversary for network analysis to introduce adversarial code. Sensor networks have a wide range of uses, and a correspondingly wide range of security requirements. By considering these three issues, we can then evaluate security mechanisms against them.

3.1 Requirements

Maté networks deployed for different purposes can have different security requirements. Almost any security mechanism has some cost associated with it, whether it be communication, memory, or CPU cycles; in the end, they cost energy, reducing the lifetime of a network. Always forcing every network to have the highest degree of security is therefore wasteful and unreasonable. A home garden soil moisture monitoring network has very different requirements than a home security system or a military network. We have identified five possible security requirements that a Maté network might have:

- **Code integrity:** in the scope of this paper, code integrity refers to end-to-end capsule and program in-

tegrity from the PC base station to a node in the network; motes can verify that a capsule originated at the PC and has not been since modified. One of the strengths of Maté is being able to quickly reprogram a network; making sure that capsules aren't corrupted or falsified prevents an adversary from installing his own program. TinySec and Active Messages provide packet but not end-to-end integrity. Capsule integrity makes no promises about the combination of capsules; an adversary could suppress the propagation certain capsules, resulting in an invalid combination and disrupting operation. Program integrity is the ability for a mote to know what combinations of capsules forming a program are valid.

- **Code confidentiality:** if an adversary knows the program running on the network, he can more easily circumvent or disrupt its purpose. Code integrity can prevent an adversary from taking control of a network, but code confidentiality can make it more difficult to counter the network's purpose.

- **Data integrity/confidentiality:** data integrity means that a mote can verify a data packet was generated from another Maté node in the network. While code confidentiality can make it more difficult for an adversary to determine the current Maté program, analyzing data can often be just as effective as looking at code. Additionally, even in the absence of an adversary trying to control or circumvent a network, data may be sensitive; even innocuous data as temperature or light readings could tell a burglar when residents are home and awake.

- **Communication Semantics** often, for a program to run correctly, certain implicit semantics must hold true. Data integrity can provide guarantees that a piece of data was generated by the network, but not when. A simple way to disrupt the intended purpose of a program can be to flood the network with replayed data packets. For the program to run correctly, it must be able to provide guarantees on the expected communication semantics (e.g., only receiving a packet once). We define communication semantics as a Maté data packet being idempotent.

- **Physical Compromise:** lastly, some networks might need to be resistant to physical mote compromise. In a home garden, this is hardly a problem, but in a home alarm system it can be. Physical mote compromise is the ability for an adversary to read the entire state of the mote and reprogram it freely. While current mote hardware makes reading state difficult, it is not impossible. We assume that only a small number of nodes can be compromised; large scale compromise is beyond the scope of this work. We discuss this further in Section 4.1.

3.2 Threats

Initial versions of Maté have several security threats:

- **Adversarial capsules** are a clear threat to a Maté network; a single instance of an adversarial capsule will virally spread through the network. Once programmed, the network can be made reprogramming-resistant (but not nonprogrammable) by saturating the network with packets, while simultaneously erasing stored data, consuming energy, and preventing operation. Capsule version numbers mean that the network can normally be reprogrammed, but an adversary could easily choose the highest version number, rendering the network reprogrammable.

- **Traffic control** allows an adversary to decide what program runs in the network. In the extreme case, putting the base station in a Faraday cage allows an adversary to completely control all traffic injected into and read from the network (by putting an adversary node in the cage). This can be used to control which capsules are installed into the network, allowing cut-and-paste attacks with combinations of capsules that result in an adversarial program.

- **Data fabrication** can be used to control a network. If Maté uses its receive handler, an adversary can cause the handler to execute on chosen or replayed data packets. For example, if the receive handler were used to configure constants such as sampling periods, data packets could cause the network to sample at useless or unmaintainable rates. Additionally, fabricated data can invalidate the results being read from the network, rendering it useless.

- **Data replay** can also be used in a similar fashion to data fabrication, although n a less precise fashion (the adversary only has the set of previously transmitted packets to use, as opposed to any packet).

3.3 Maté Cryptanalysis Weaknesses

Maté's viral code propagation has very distinctive network traffic patterns and characteristics that could possibly be utilized by an adversary in cryptanalysis. These include:

- **Repetition:** A given capsule can be transmitted many times in a network; each mote transmitting the capsule sends an identical packet. Adversaries can use this repetition in cryptanalysis efforts.

- **Weak cleartext:** The AM and capsule headers in a capsule packet are very weak clear text; they comprise a tiny slice of the possible binary space. For example, the AM destination, AM type, and packet length are fixed, the capsule type is one of a narrow set of values, version numbers are monotonically increasing values (which could naively be merely an increment of a previous one), and the capsule options field has very low entropy, and code is also lower than perfect entropy. An adversary could utilize these characteristics to launch cryptanalytic attacks.

- **Traffic rates:** By analyzing the rate of traffic being sent, an adversary can learn facts about its contents. For example, if motes in the network change their network behavior in a wave-like pattern, one can intuit that a new capsule is possibly propagating through the network. Similarly, by partitioning the network, an adversary can determine if a packet is sent at a standard interval independent of other traffic (a timer), or in response to receiving a packet (ad-hoc routing).

3.4 TinySec and Symmetric Keys

TinySec uses symmetric key cryptography to provide message integrity and confidentiality. Randomized initialization vectors provide a limited form of confidentiality for repeated messages. The RC5 CBC used is very resistant to cryptanalysis [20, 10, 2, 19]. As not all of a packet (e.g. AM type) is encrypted in TinySec, however, an adversary can easily determine which packets are capsules. TinySec packets can be used in cut-and-paste attacks against Maté to install faulty or invalid programs.

A single shared key in the network provides relatively inexpensive (in terms of CPU and storage) message confidentiality and integrity. However, a very determined adversary (such as in a wartime situation) can extract the key using a variety of indirect techniques, including determining CPU behavior from power traces[11, 12, 14]. Alternatively, the adversary can just take the chip to a suitably equipped fabrication facility to extract the key from memory. Both of these assume that the memory is hardware-protected; while this is currently true, there is no assurance it will continue to be so on future platforms. In the absence of hardware access controls to memory, leaning the symmetric key is trivial if one can physically access a mote.

4. DESIGN

We present several mechanisms that Maté can use to meet a user's security requirements. We have implemented some of them, while others are future possible inclusions; implementation has been postponed due to the significant changes they could bring to the VM programming model.

4.1 Trusted Computing Base

In all of our security scenarios, we consider the PC base station part of the TCB. Compromise of the PC would allow an adversary to freely reprogram the network, erase or modify data, and in all respects make the network useless.

If the security requirements of a network include resistance to physical mote compromise, then individual motes cannot be considered part of the TCB. While a network can be resistant to a small number of motes being compromised, large compromises are clearly difficult to counter. If every mote in the network is invisibly operating for the adversary (physical compromise assumes invisibility), there is little an administrator can do.

If the network is not required to be resistant to physical compromise, than motes programmed by the network administrator can be considered part of the TCB; foreign motes, however, are not.

4.2 TinySec

In the absence of physical mote or key compromise, TinySec can provide code and data confidentiality as well as acceptable degrees of data integrity. By itself, it cannot provide complete code integrity, as a capsule may be corrupted in memory on a mote. TinySec cannot provide program integrity; an adversary could still use cut and paste attacks.

4.3 Code Signing

Code integrity requires end-to-end capsule integrity, which can be achieved by including a cryptographic hash of the capsule. If the hash is plaintext, however, then code integrity requires code confidentiality and resistance to physical compromise. Otherwise, knowledge of the hash function could allow an adversary to generate his own capsules and sign them.

Maté's viral code propagation means that if an adversary can install a capsule on a single uncompromised mote, it will spread through the entire network. To protect the network from this, capsules must be digitally signed by a trusted PC. Compromising motes in the network cannot compromise the signing process; asymmetric cryptography must be used. Motes, however, must be able to verify the signature. A network is deployed with a public key. When a PC generates a capsule, it signs it using a private key. Before a mote installs a capsule, it checks the signature. AM-level headers are not signed; as long as different AM types use different signatures, cut and paste attacks cannot be used. We use the bottom 64 bits of an MD5 hash in the signature.

We present two possible implementations, one unpromising and one promising. Public key cryptography can be used to sign code capsules, using RSA with carefully selected constants to minimize mote-side computation. Key sizes make this approach unfeasible. We therefore examine variants of BiBa [16] signing as a low-overhead, concise alternative.

4.3.1 Public Key Signing

Public key operations are expensive in storage and CPU. We can use the symmetry of RSA to make decryption the easier of the two operations; the mote can use an e of 3 or 17, making decryption comprising a few multiplications and a modulus. Additionally, the mote only needs to check the signature if it thinks it would use the capsule or version vector (based on its contents). This does allow an adversary to launch an attack by sending bogus high version number capsules, whose signatures a mote will keep checking. A realistic high-security public key (2048 bits) would consume just under ten percent of a current mote's memory resources; while a heavy requirement, it is impossible.

Far more problematic, however, is the required packet size; signatures are as long as the public key, greatly outweighing the size of the data. Additionally, to use a 2048 bit key, however, the clear text must be of suitable length for the modulus. Using an e of 3 would require a plaintext of 750 bits (95 bytes). A larger e is preferable, but can increase the computational requirements for the mote.

Using PKC, capsule integrity can be provided in the presence of in-memory capsule corruption and physically compromised motes. However, its memory and communication requirements are unfeasible. We have implemented a very limited version of RSA (a 64-bit key) as a proof of concept for this technique, but do not plan to extend it further unless the bandwidth provided by sensor networks increases significantly.

4.3.2 BiBa Signing

BiBa is a signature mechanism whose signatures that are very easy to verify [16]. A signature consists of a set of values V_i (e.g. 64-bit numbers) and a counter. To verify the signature, the verifier takes a hash of the message, H_m, and adds the counter. The verifier computes the hash of each V_i using H_m as an additional input. The signature is valid if each V_i hashes to the same value. BiBa has several variants and options (such as having multiple sets, each which hashes to a value), which we do not discuss for sake of brevity [15, 18].

The values $V_0 \ldots V_n$ are the private key; the verifier has a

public key that allows easy verification that V_i is authentic. As a sender reveals more V_i, an adversary has a greater chance of being able to forge a signature. This raises the problem of key distribution.

4.3.2.1 Private Key BiBa.

We can circumvent the key distribution problem by using a symmetric key. Each verifier has the full set of signing values; instead of including values in a signature, the sender sends which signing value is being used (an index into an array). In addition to not revealing the values to an adversary, this allows a more concise representation; 64-bit signing values can be represented as 10-bit indexes.

The BiBa private-key signature then takes this form, adding five bytes to a message:

```
typedef struct BombillaBiBaSignature {
  uint32_t indexes;
  uint8_t counter;
} BombillaBiBaSignature;
```

The 1024 64-bit signing values are installed on a mote at programming time, and are stored in program memory. The implementation uses an MD4 hash function, and uses a 3-way collision; the three ten bit indexes are stored in the 32 bit field. Using a symmetric key allows a PC to reprogram the network many times safely. However, it makes the network vulnerable to physical compromise.

4.3.2.2 BiBa with One-way Signing Value Chains.

This is currently unimplemented. -pal

Physical compromise can be countered by using the standard BiBa scheme: instead of storing the signing values, the mote stores value authenticators, the hash results of signing values. Using a trapdoor function prevents an adversary from deducing signing values. However, by revealing signing values in plaintext, a sender allows an adversary to generate a dictionary of values that can then be used to forge signatures. Perrig et al. comment that BiBa's security holds as long as less than roughly 10% of the values are revealed (given setting the other constants properly). Additionally, the sender must transmit entire values instead of indexes (i.e., 64 instead of 10 bits), greatly increasing the signature overhead.

However, using one-way chains, BiBa can be implemented using a public key in a manner that doesn't require periodically distributing a new public key. Each signing value can only be used once; a signature contains the counter and two signing values, each with a generation number. The generation number denotes which element in a one-way chain the value represents. The public key begins as the set of 0-generation values; given a signing value, the mote can compute the hash chain to the public, known value to verify it. A mote only accepts signing values of later generations that hash forward to the public key. Because each signing value can only be used once, using two signing values provides adequate security.

This forward-chaining does have one vulnerability; an adversary can use a traffic control attack to separate a set of motes from the network, listen for a number of signing values, then sign its own adversarial capsule, infecting the separate motes. The rest of the network, however, will not accept the signature, as it will use signing values that have already expired. Of course, an adversary could separate the entire network (by quarantining the base station); however, if the administrator is unaware that the past n reprogrammings haven't propagated to the network, that's a wholly different problem.

Assuming chains of maximum length 2^{16}, forward-chaining BiBa would be a 24-byte overhead on each packet; 16 bytes for the signing values, four bytes for their generations, three bytes (20 bits) to state which signing values they are, and one byte for the counter. Because the signing values are invalidated as soon as they are revealed, the best chance an adversary would have to use them would be to try to use them on a distant part of the network before they propagate. More signing values could be added to improve security, at the cost of roughly 11 bytes per value.

4.4 Program Vectors

If an adversary has complete control over traffic in the network, code integrity cannot be provided by merely signing capsules. The adversary can control which capsules make their way into the network, constructing combinations that provide incorrect data (e.g., by not letting a new data filter be installed). The adversary can also use cut-and-paste attacks to generate possibly adversarial programs.

Maté already has capsule version vectors for code propagation. By making a small modification to the vector packet format, we added support for **program vectors**, which specify valid capsule combinations. These version vectors originate at a PC and are signed just as capsules are. As all version numbers are monotonically increasing, each vector field must be monotonically increasing as well (otherwise, it could become impossible to reprogram a network as the required capsule will not be installed). When a mote receives a new program vector, it stores it and continues to run the current program. As soon as it hears one of the capsules required for the new program, it halts execution until it has all of them.

4.5 Data Clusters

This is currently unimplemented. -pal

Capsules originate from a single point, a powerful PC, allowing the use of public key cryptography to provide integrity. In a sensor network, however, data originates from every node and must be received by other nodes. Even if data integrity and confidentiality are provided, an adversary can disrupt execution of the network through replay attacks.

Data packets sent using Maté's built-in ad-hoc routing are not considered by the VM; once the instruction is issued, the ad-hoc subsystem routes the packet to a base station. Replay suppression and data inconsistencies can therefore be considered in a centralized place. From Maté's perspective, the difficult issue is not there packets, instead VM data packets that trigger receive handlers, as they require distributed countermeasures.

μTESLA implements authenticated broadcast using per-epoch keys and a loose form of time synchronization [17]. However, this mechanism only protects a static network in which key sequences are securely bootstrapped; otherwise, one could save the key/data sequence from one region of the network and replay it in another.

Communication semantics requires that Maté data packets be idempotent. Capsule idempotency is implemented through version numbers. A global version number is unfeasible for a sensor network, as it would require a centralized

mechanism that would be fragile to traffic control attacks.

To protect Maté from data replay, we use a fixed-size neighbor cache and sequence numbers. Every Maté data packet (sent with the send capsule) has the receive capsule version number in it. The neighbor cache is initialized whenever a new receive capsule is installed; the mote installs the first eight motes it hears transmit a message with the current receive version. It keeps a 32-bit sequence number for each mote in the cache, and only accepts packets if they are both the current receive handler and have an increasing sequence number. Assuming nonchargeable battery power, the number of packets a mote can send in its lifetime is well below 2^{32}, so there is no chance that the sequence number will wrap around.

Use of dynamic data neighbor clusters provides data replay protection for mobile networks (although it requires a new capsule to generate new clusters). The rate of mobility is constrained, however, by the rate of propagation; it cannot support high mobility in which clusters change very frequently. Additionally, the formation of clusters requires feedback for any sort of directly addressed packet; it is possible that mote A has mote B in its cluster, but not vice-versa. Mote B will therefore never receive packets from mote A.

Data clusters do have vulnerabilities. For example, an adversary can use a one-for-one replay of data packets from a mote in different areas of the network; many nodes could place that mote in their neighbor caches, making the logical topology have a much greater reach than the physical topology. This is similar to a combination of the wormhole and Sybil attacks presented by Karlof et al. [9], except that it disrupts data instead of routing. The problem posed by this attack is that it can be entirely indistinguishable from the very spotty and unpredictable behavior of a real-world network. The one warning sign could be if a mote hears its own packet replayed, but countermeasures to this attack cannot assume this will be the case.

5. EVALUATION

We describe how each of the security requirements outlined in Section 3.1 can be satisfied with the mechanisms we have proposed. Physical compromise makes some goals impossible, specifically confidentiality; for example, if a mote's memory can be read, achieving code confidentiality is problematic. Maté receive capsule data confidentiality is similarly difficult to provide, although PKC can be used for centrally collected data. We then consider the overhead these mechanisms pose.

5.1 Requirements

We consider the possible security requirements in turn, with physical compromise being considered as an additional circumstance for each of the others.

5.1.1 Code Integrity

Code integrity can be obtained by adding a MAC to each capsule. Unless an adversary learns the MAC key, he cannot generate new capsules. Packet-level (e.g. TinySec) MACs do not protect the network from internal corruption. In the presence of physical compromise, the key to generate a MAC cannot reside on a mote. Using public key signatures provides capsule integrity even in the presence of mote compromise, and signed program vectors prevent cut-and-paste attacks. In combination, they provide code integrity.

5.1.2 Code Confidentiality

Code confidentiality can be provided by TinySec. A mote must be able to access the plaintext of the code in order to execute it; for this reason, code confidentiality is unfeasible in the presence of mote compromise mechanisms.

5.1.3 Data Integrity/Confidentiality

Data integrity and confidentiality can be provided by TinySec's MAC. In the presence of mote compromise, integrity and confidentiality can be provided for routed data with PKC, but Maté data packets are vulnerable for the same reasons as code integrity.

5.1.4 Communication Semantics

Communication semantics require that a mote only receive a sent Maté packet once. By defining a logical topology, the motes in a Maté network can keep state on each of their neighbors, making data packets idempotent. The idempotency of routed data packets can be handled by the central collection PC, with its filtering and processing capabilities. In Maté data packets, idempotency is not compromised by the presence of compromised nodes. However, if the network cannot provide data integrity, than an adversary can inject new packets into the system, disrupting results.

5.2 Overhead

We present the overhead of each of the proposed mechanisms.

5.2.1 Code Signing:

Code signing poses both a message length and a CPU overhead (for signature verification). Private-key BiBa imposes a five byte overhead on a packet. Assuming chains of maximum length 2^{16}, forward-chaining BiBa would be a 24-byte overhead on each packet; 16 bytes for the signing values, four bytes for their generations, three bytes (20 bits) to state which signing values they are, and one byte for the counter. Because the signing values are invalidated as soon as they are revealed, the best chance an adversary would have to use them would be to try to use them on a distant part of the network before they propagate. More signing values could be added to improve security, at the cost of roughly 11 bytes per value.

However, the RAM constraints on a mote make a strong version of BiBa impossible to efficiently implement; the key (public or private) is too large to store in RAM. Instead, it must reside on the EEPROM; reading signing values from the EEPROM would greatly increase the time it takes to verify a signature. A careful analysis of the threat probabilities (based on the one-use value and timing vulnerability) could allow the algorithm to use fewer than the recommended 1024 signing values, reducing the memory requirements.

5.2.2 Program Vectors:

Program vectors are sent using the same timing finite state machine as version vectors. However, unlike version vectors, whose timer is reset for each new capsule, program vectors only reset their timer once per combination of capsules. If a program runs in a Maté network for eight hours, the timing decay will cause it to send 241 program vectors, an average of one every two minutes.

5.2.3 Data Clusters:

Implementing data clusters would require a small amount of storage for the local neighborhood, on the order of fifty bytes. More significant, however, are the limitations it creates on the operation of a network; the topology of the network is limited to degree eight.

6. DISCUSSION

The primary overhead in the Maté security mechanisms is the increased packet lengths; given the energy cost of communication, this is a high cost. We have not made a detailed analysis of the required minimum sizes of these security-related fields, instead implementing them at safely large sizes. The amount of data included is an obvious parameter than can be tuned between requirements of efficiency and security.

This concern stems from a characteristic of sensor networks that distinguishes them from other domains. In mobile systems, for example, omnipresent public key cryptography is unfeasible due to the computational requirements, while computing hashes and symmetric cryptography is are readily available. In contrast, in a sensor network these latter mechanisms are limited not by their CPU requirements as much as their data requirements. Adding a four byte hash onto every message is a significant cost. However, many of the vulnerabilities requiring large hash sizes in more powerful systems, such as birthday attacks, are not feasible in sensor networks due to their energy limitations; motes can't receive the same volume of packets.

The major problem none of our mechanisms can solve is confidentiality in the presence of compromised nodes. Additionally, while data clusters bound the damage that data replay and compromised nodes can cause, a network can still be disrupted. Unfortunately, prior work in distributed system fault tolerance is far too communication-heavy for sensor networks, even when intended to be practical [3]. This suggests that perhaps compromise detection is preferable to compromise prevention.

The many-to-one relationship between a base station PC and sensor network motes facilitates the utilization of public key cryptography for communication between the two domains. Currently, the CPU and storage requirements of public key encryption make it unfeasible for reasonably sized keys. Small data payloads also pose a problem for PKC operations; for example, signatures must be as long as the key (≈300 bytes)..

7. CONCLUSION

Sensor networks can be used in a wide range of domains, with differing security requirements. Maté provides a rapid and autonomous programming interface to a network, but raises significant security vulnerabilities. TinySec can be used to solve some of these vulnerabilities, but others require mechanisms in Maté itself. TinySec is not always applicable; there are use cases in which its confidentiality is not necessary, but end-to-end signing is a core requisite for Maté capsules regardless of network security. Code signing in the presence of mote compromise requires some form of public-key signing; while some current public key signing systems such as BiBa are feasible, they have require a lot of storage. Efficient public-key signing is an important future research goal for Maté networks.

8. REFERENCES

[1] Atmel Corporation. *ATmega128(L) Preliminary Summary*, 2002.

[2] A. Biryukov and E. Kushilevitz. Improved cryptanalysis of RC5. *Lecture Notes in Computer Science*, 1403:85–??, 1998.

[3] Castro and Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[4] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. Planetp: Infrastructure support for p2p information sharing.

[5] D. Estrin, R. Govindan, J. S. Heidemann, and S. Kumar. Next century challenges: Scalable coordination in sensor networks. In *Mobile Computing and Networking*, pages 263–270, 1999.

[6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000. TinyOS is available at http://webs.cs.berkeley.edu.

[7] S. T. B. N. in Small Tech. http://www.smalltimes.com.

[8] C. Karlof, N. Sastry, , and D. Wagner. TinySec: Security for TinyOS, 2002. Presentation given at NEST group meeting, 11/21/2002.

[9] C. Karlof and D. Wagner. Secure routing in wireless sensor networks: Attacks and countermeasures.

[10] L. R. Knudsen and W. Meier. Improved differential attacks on RC5. *Lecture Notes in Computer Science*, 1109:216–??, 1996.

[11] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. *Lecture Notes in Computer Science*, 1666:388–397, 1999.

[12] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. *Lecture Notes in Computer Science*, 1109:104–113, 1996.

[13] P. Levis and D. Culler. Maté: a Virtual Machine for Tiny Networked Sensors. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2002.

[14] T. S. Messerges, E. A. Dabbish, and R. H. Sloan. Investigations of power analysis attacks on smartcards. pages 151–162.

[15] M. Mitzenmacher and A. Perrig. Bounds and improvements for biba signature schemes.

[16] A. Perrig. The biba one-time signature and broadcast authentication protocol. In *ACM Conference on Computer and Communications Security*, pages 28–37, 2001.

[17] A. Perrig, R. Szewczyk, V. Wen, D. Cullar, and J. Tygar. Spins: Security protocols for sensor networks, 2001.

[18] L. Reyzin and N. Reyzin. Better than biba: Short one-time signatures with fast signing and verifying. Technical Report 2002/014, 2002.

[19] B. Schneier. *Applied Cryptography, Second Edition*. 1996.

[20] T. Shimoyama and K. Takeuchi. Correlation attack to the block cipher RC5 and the simplified variants of RC6. In *Proceedings AES3*, 2001.

[21] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the International Symposium on Computer Architecture*, pages 256–266, 1992.

[22] B. Warneke, M. Last, B. Liebowitz, and K. Pister. Smart dust: Communicating with a cubic-millimeter computer. *IEEE Computer Magazine*, pages 44–51, January 2001.