Using FIRE & ICE for Detecting and Recovering Compromised Nodes in Sensor Networks

Arvind Seshadri, Mark Luk, Adrian Perrig,
Leendert van Doorn, Pradeep Khosla

December 2004
CMU-CS-04-187

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper presents a suite of protocols called FIRE (Forgery-resilient Intrusion detection, Recovery, and Establishments of keys), for detecting and recovering compromised nodes in sensor networks. FIRE consists of two protocols: an intrusion detection and code update protocol, and a cryptographic key update protocol. In concert, the FIRE protocols enable us to design a sensor network that can always detect compromised nodes (no false negatives), and either repair them through code updates and set up new cryptographic keys, or revoke the compromised nodes from the network.
The FIRE protocols are based on ICE (Indisputable Code Execution), a mechanism providing externally verifiable code execution on off-the-shelf sensor nodes. ICE gives the following two properties: 1) the locations in memory from where the code is currently executing on a sensor node, matches memory locations being verified and 2) the memory contents being verified are correct. Together, these two properties guarantee that the code currently executing on the sensor node is correct.
The FIRE protocols represent a significant step towards designing secure sensor networks. As far as we are aware, there are no techniques for intrusion detection in adhoc and sensor networks that do make any false negative claims. Also, we do not know of any existing techniques that can automatically recover compromised sensor nodes.
We present an implementation of our FIRE protocols and ICE on current off-the-shelf sensor devices.

This research was supported in part by the CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, and grant CAREER CNS-0347807 from NSF, and by gifts from Bosch and IBM. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of ARO, Bosch, Carnegie Mellon University, IBM, NSF, or the U.S. Government or any of its agencies.

Keywords: intrusion detection, code update, key update, self-verifying code, sensor network

1 Introduction

Sensor networks are expected to be deployed in the near future in a variety of safety-critical applications such as critical infrastructure protection and surveillance, military applications, fire and burglar alarm systems, home and office automation, inventory control systems, and many medical applications such as patient monitoring. Sensor nodes in a sensor network typically run identical software. Vulnerabilities in the sensor node software, like buffer overflows, leave all sensor nodes vulnerable to remote exploit. An attacker can exploit a vulnerability to inject and execute arbitrary code on the sensor nodes, steal their cryptographic keys, and possibly also compromise the privacy and safety of people. Security is especially challenging to achieve in this setting, due to the severe resource constraints of limited memory, computation, battery energy, and wireless communication bandwidth in current sensor network technology.

This paper presents a suite of protocols, called FIRE (Forgery-resilient Intrusion detection, Recovery, and Establishments of keys), for detecting and recovering from intrusion in sensor networks. An *intrusion* is defined as the process of a remote attacker compromising a sensor node using software vulnerabilities such as a buffer overflow. We design a novel approach for intrusion detection in sensor networks that does not make any false negative claims,[1] i.e., when our technique claims that a node is uncompromised, the node is indeed uncompromised. Conversely, if a node is found compromised, it is either compromised, or uncompromised but under a denial of service (DOS) attack. However, an uncompromised node that is found compromised due to a DOS attack, may be found to be uncompromised after the DOS attack ends. Our intrusion detection algorithm uses a completely new technique. As we discuss in more detail in our related work section, previous intrusion detection approaches in wireless networks utilize wireless monitoring and heuristics to detect misuse or anomalous behavior. In these systems, neighboring nodes monitor packets sent by a node, and raise alarms if the node misbehaves. Such approaches unfortunately are susceptible to slander and framing attacks, and exhibit false positive as well as false negative detections.

Once a node is found to be under the control of an attacker, it was so far an open challenge how to regain control of the node and how to set up new cryptographic keys without human intervention. To the best of our knowledge, we present the first protocols for secure code updates and secure key establishment after node compromise in sensor networks. Our secure code update mechanism securely patches a sensor node. By "secure"we mean that a verifier obtains a firm guarantee that the patch was correctly installed and that the code image of the node, after application of the patch, is correct. Our secure key establishment mechanism enables a trusted node to re-establish a secret key with a node that was compromised. Our approach to key establishment is immune to man-in-the-middle attacks without requiring any shared secrets. Moreover, an eavesdropper does not learn the established secret key. The results we present in this paper appear even more surprising since we assume commodity sensor nodes (i.e., no special hardware required). However, we do assume that the attacker only has remote access to the sensor network, i.e., the attacker is not physically present in the proximity, but communicates with the sensor nodes through a network. Our techniques can detect and recover nodes even if an attacker compromises an arbitrary number of sensor nodes, uploads arbitrary code into the node, and where nodes can arbitrarily collude.

All our protocols are based on a new mechanism: ICE, which stands for Indisputable Code Execution. ICE is a request-response protocol between the verifier and the device. The verifier does not have

[1]We define a *positive* as a sensor node that is compromised, and a *negative* as a node that is uncompromised.

physical access to the device's CPU, bus, or memory, but can only communicate over a wireless link with the device. ICE verifies exactly what code is executing on the device at the time the verification is performed. The verifier sends a request to the device. The device computes a response to the verifier's request using a verification function (hereafter called the ICE verification function), and returns the response to the verifier. A correct response from the device guarantees to the verifier that two properties hold on the device. First, the location, in memory, of the code currently executing on the device, matches the location, in memory, of the content we are verifying. Second, the memory contents being verified are correct. Taken together, these two properties assure the verifier that the code it expected to execute on the device, at the time of verification, did indeed execute on the device. A correct response from the device should guarantee that the two properties mentioned above hold even if the attacker controls the node before ICE runs and make arbitrary changes to the memory content.

We present an implementation of FIRE & ICE, Our implementation is based on the Telos sensor nodes, the most recent sensor platform of the Berkeley mote family [12].

Outline In Section 2, we present the problem definition, and describe the sensor network architecture, assumptions, and the attacker model. Section 3 describes the ICE mechanism. In Section 4, we describe FIRE as well as our implementation and evaluation of the FIRE protocol suite. Section 5 discusses related work, and Section 6 concludes.

2 Problem Definition, Assumptions and Attacker Model

We first state our assumptions about the sensor network architecture in Section 2.1. Section 2.2 discusses our attacker model. In Section 2.3, we describe the problem of detecting and repairing remote intrusions in sensor networks.

2.1 Sensor Network Assumptions

We assume a wireless sensor network consisting of one or multiple base stations and several sensor nodes. The sensor nodes communicate among themselves and the base station using a wireless network. The communication between the base station and sensor nodes can be single-hop or multi-hop.

The base station is the gateway between the sensor network and the outside world. Other sensor networks or computers on the world-wide Internet can send network packets to the sensor nodes through the base station. Every sensor node and the base station has a unique identifier, hereafter referred to as *node ID* or *base station ID*.

To authenticate messages between sensor nodes and the base station, we assume, for simplicity, that a public-key infrastructure is set up, where each sensor node knows the authentic public key of the base station (we assume that the base station is the Certification Authority (CA) of the network). Malan et al. have recently shown that public-key cryptography takes on the order of tens of seconds on current sensor nodes [23], which is justifiable for a small number of operations. We could also assume pairwise shared keys between the base station and sensor nodes, and use the SPINS infrastructure to set up additional keys [26]. We assume that the base station is immune against remote attacks that inject and run arbitrary code on the base station or steal the cryptographic keys in the base station. This assumption is commonly made in secure sensor networks, since compromise of the base station implies compromise of the entire network.

We further assume that each sensor node has a few bytes of Read-Only Memory (ROM). The ROM stores the node ID of the sensor node and base station's public key. By keeping a sensor node's node ID in the ROM, we prevent impersonation attacks where an attacker changes the node ID of a node to impersonate another node, for example the Sybil attack [8]. The base station's public key is used by the sensor nodes to authenticate packets from the base station. Storing the base station's public key in ROM prevents an attacker from changing that key if a node is compromised.

We also assume that the code that implements FIRE & ICE, being small in size (approximately 3-4 KB), can be carefully written to be free from software vulnerabilities like buffer overflows.

2.2 Attacker Model

In this paper, we study all remote attacks that an attacker can launch against the sensor nodes in a sensor network. A remote attacker exploits vulnerabilities, like buffer overflows, in the software running on the sensor nodes to compromise the sensor nodes. Once a node is compromised, the attacker has full control. That is, the attacker can inject and run arbitrary code, and steal cryptographic keys. Malicious nodes controlled by the attacker can collude. We assume that the attacker does not introduce its own powerful hardware like laptop computers into the sensor network to impersonate sensor nodes. Introducing new hardware into a sensor network requires the attacker to be physically present which translates to a substantially more determined attacker. In many physically secure sensor networks like those in nuclear power plants or in military environments, the attacker will not be able to introduce its own hardware into the network.

In our future work, we will consider an attacker who is present at the sensor network, allowing it to introduce its own malicious and computationally powerful sensor nodes.

2.3 Problem Definition

We consider the setting of a sensor node that has a software vulnerability in its code. An attacker can exploit the vulnerability to compromise the sensor node. After compromising the sensor node the attacker can read out the memory contents or inject malicious code into the node.

When a vulnerability is discovered in the sensor node software, the base station has to first detect which sensor nodes in the network have been compromised by an attacker. For uncompromised nodes, their code has to be updated to remove the vulnerability. The compromised nodes either have to be repaired or be blacklisted by the base station.

To repair compromised nodes, first, any malicious code or changes made by the attacker have to be removed from the memory of the sensor node. Then, the code running on the sensor needs to be updated to remove the software vulnerability. All repair needs to be done in the presence of malicious code that may prevent the repair from happening. For example, if the base station sends a software patch to a compromised sensor node, malicious code running on the node may fake the application of the patch.

New cryptographic keys have to be established between the base station and the all sensor nodes. Even uncompromised nodes need new cryptographic keys because an attacker could have compromised a node, read out the cryptographic keys, and then undone all changes to make it appear as though the node were not compromised. The protocol used to establish new cryptographic keys cannot be based on the assumption of the existence of any shared secrets between the base station and sensor nodes. All shared secrets might have been compromised. Even without shared secrets, the cryptographic key establishment protocol has to be immune to eavesdropping and man-in-the-middle attacks.

3 ICE: Indisputable Code Execution

In this section, we first describe the indisputable code execution mechanism and show how self-verifying code can be used to achieve indisputable code execution. Section 3.2 shows attacks against self-verifying code to spoof the existence of the indisputable code execution property and the properties our self-verifying code (hereafter called the ICE verification function) has that prevent these attacks. In Section 3.3, we describe the design of the ICE verification function to achieve its required properties. Finally, Section 3.4 discusses the implementation of the ICE verification function on the Telos sensor nodes.

3.1 Indisputable Code Execution

We consider the model where a verifier wants to verify what code is executing on a device, when the verification is performed on the device. However, the verifier does not have physical access to the device's CPU, bus, or memory, but can only communicate with the device over a network link. The verifier knows the exact hardware configuration of the device, and can command the device to run a self-verifying checksum function, called the ICE verification function. In this model, the verifier sends a challenge to the device, asking the device to use the ICE verification function to compute and return a checksum over the contents of memory from which the ICE verification function runs. If the ICE verification function that runs on the device is correct and the ICE verification function is running from the correct location in memory, the device responds with the correct checksum within a pre-specified time period; if the ICE verification function is incorrect or running from a different location in memory than that expected by the verifier, either the checksum will be incorrect with overwhelming probability, or the device will respond after the pre-specified time period (since the ICE verification function is designed to execute slower if the ICE verification function code is different or it runs from a different location in memory). This is the same setting as previous research on this topic assumes [16, 29]. Figure 1 shows an example of a verifier that verifies the code executing on a device.

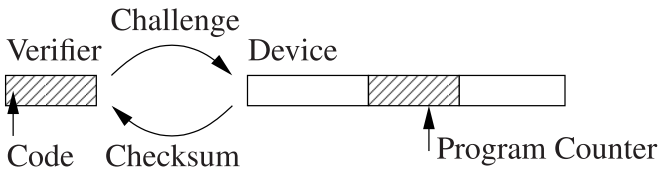

Figure 1. Setting of Indisputable Code Execution, a verifier wants to ensure that a device is indeed executing a certain piece of code. The verifier knows the hardware architecture of the device and the value of the piece of code it wants to verify. The device's memory contains the code the verifier wants to verify. The verifier sends a challenge and only receives the correct checksum within a bounded time period if the device is indeed executing the correct code.

The property of ICE (Indisputable Code Execution) is that it "freezes" the code on the device, such that the verifier obtains assurance about what code is currently running. As we show in Section 4, ICE is a powerful primitive that enables a wide variety of higher-level security mechanisms, such as secure verifiable code updates, secure key establishment, and intrusion detection.

We use self-verifying code to implement ICE. We define *self-verifying code* as a sequence of instructions, that compute a checksum over themselves in a way that the checksum would be wrong or the computation would be slower if the sequence of instructions were modified.

3.2 Attacks Against Self-Verifying Code

We now describe potential attacks against any self-verifying code and intuitions on how we design our defenses.

Pre-computation and replay attacks. An attacker can compute the checksum over the memory region containing the ICE verification function, before making changes to the ICE verification function. Later when the verifier asks the device to compute and return the checksum, the device returns the pre-computed value. To prevent this attack, the verifier sends the device a random challenge along with every verification request. The checksum computed by the device is a function of this challenge. The challenge sent by the verifier is sufficiently long to prevent replay attacks when the attacker stores previously observed challenge-checksum pairs.

Data substitution attacks. The ICE verification function makes multiple linear passes over memory region from which it runs and iteratively computes the checksum. An attacker can change a some bytes of the ICE verification function and keep the original values at a different location in memory. When the ICE verification function tries to read from the memory locations the attacker changed, the attacker diverts the read to the locations in memory where it stored the original values. The attacker has to insert an `if` statement before the instruction in the ICE verification function that reads from memory to check when the read goes to one of the locations it changed. Inserting the extra `if` statement slows down the computation of the checksum. This slowdown will be detected by the verifier when it does not receive the checksum from the device within the correct time. However, the attacker can make one optimization to reduce the overhead of the extra `if` statement. We unroll the loop of the ICE verification function. Thus, the body of the ICE verification function is composed of several instances of the basic loop. Since the ICE verification function makes linear passes over memory, the attacker can predict in advance which loop instances will access the memory locations it modified. Then, it can insert the `if` statements in those loop instances alone. To force the attacker to insert `if` statements into every instance of the loop, we unroll the ICE verification function loop so that the number of instances of unrolled loop and the size of the memory region over which the checksum is computed are coprime. This ensures that the same memory location will be accessed by different loop instances during different passes over the memory region. So, the attacker is forced to insert `if` statements into every loop instance.

Memory copy attacks. Since we only want to verify the code that is currently executing on a device and that code only constitutes a small part of the full memory contents of the device, we are faced with two copy attacks: either the correct code is copied to another location in memory and malicious code is executing at the location of the correct code (Figure 2), or the correct code resides at the correct memory location and the malicious code is executing at another location of memory (Figure 3). It is clear that we need to prevent both attacks to have self-verifying code. To prevent the first attack, we need to ensure that the contents that we compute the checksum over are fetched from the correct address locations in memory. To prevent the second attack, we need to ensure that the program counter is pointing to the correct memory addresses. A third attack is that both the correct code and the malicious code are at different memory locations. It is clear that either of the countermeasures that prevent the first or second copy attack also prevent the third attack.

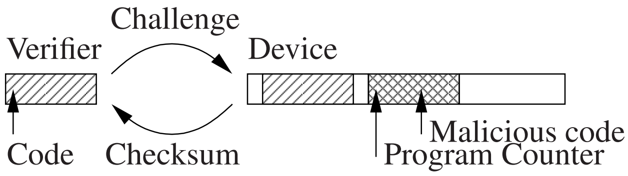

Figure 2. In this attack, the correct code resides at a different memory location, and the attacker executes malicious code at the correct memory location, computing the memory checksum over the correct code.

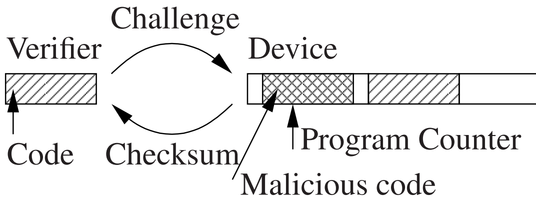

Figure 3. In this attack, the correct code resides at the correct memory location, but the attacker executes malicious code at a different memory location, computing the memory checksum over the correct code.

Loop counter modification attack. The attacker that has modified a certain portion of the memory being checked modifies the termination value of loop counter so that the ICE verification function runs until it reaches the beginning of the modified memory region. Once the loop is exited the attacker forges the checksum over the memory it modified. After that, the attacker jumps back to the legitimate copy of ICE verification function and runs it until completion. Although the attacker's forgery of the checksum has incurs a time overhead, this overhead would be limited to a small portion of the memory content under examination. The result is that the time overhead might be too small to be detected by the verifier. To prevent this attack, we incorporate the termination value of the loop counter into the checksum to catch modifications to this value.

Since the ICE verification function depends on timing, many attacks attempt to speed up the checksum computation. This allows the attacker to run a malicious verification function and use the time gained by speeding up the checksum computation to forge the correct checksum. As long as the correct checksum is returned to the verifier by a certain time, the verifier would consider this node as uncompromised. Three such attacks leveraging timing is presented below.

Computing checksum out-of-order attack. The verification function makes one or more linear passes over the memory region for which it computes the checksum. The attacker knows exactly how many times a given memory location is accessed during the computation of the checksum, thus it may compute the checksum contributions all at once without performing the iterations. This would enable the attacker to save time, and in conjunction with the memory copy attacks allow the attacker to return the correct checksum within the allocated time. Making the verification function non-associative prevents the attacker from making shortcuts in the computation.

Optimized implementation attack. The attacker may decrease the execution time of the ICE verification function by optimizing the code, which allows the attacker to use the time gained to forge the checksum, without being detected. Similar to previous research in this area [16, 29], we need to show that the code cannot be further optimized. As previously pointed out, we can use automated tools to either exhaustively find the most efficient implementation [11], or to use theorem proving techniques to show that a given code fragment is optimal [15]. In any case, our goal is to keep the code exceedingly simple to facilitate manual inspection and the use of these tools.

Multiple colluding devices attack. Another way to speed up execution is by leveraging multiple devices to compute the checksum in parallel. Multiple devices can collude to compute different ranges in the ICE verification function loop and combine their results to get the final checksum. To prevent this attack, we want to make the verification function non-parallelizable to force sequential execution.

3.3 Design of ICE

The key idea in ICE is that the ICE verification function computes a checksum over its own instruction sequence and return the checksum to the verifier within a certain time period of time. We now discuss what primitive we use to generate the fingerprint of memory.

As mentioned in Section 3.2, the checksum computation to be resistant to pre-computation and reply attacks. This requirement rules out using a cyrptographic hash function. We could use a cryptographic message authentication code (MAC), like HMAC [5]. However, MAC functions have much stronger properties than we require. MACs are designed to resist the MAC forgery attack. In this attack, the attacker has observes the MAC values for a number of different inputs. All MAC values are computed using the same key. The attacker then tries to generate a MAC for an unknown input, under the *same* key, using the input-MAC pairs it has observed. In our setting, the verifier sends a random challenge to the device along with each verification request. The device uses the random challenge as the key to generate the memory fingerprint. Since the key changes every time, the MAC forgery attack is not relevant in our setting.

We use a simple checksum function to generate a fingerprint of memory. The checksum function uses the random challenge sent by the verifier to seed a pseudorandom number generator (PRG) and to initialize the checksum variable. The output of the PRG is incorporated into the checksum during each iteration of the checksum function. The input used to compute the checksum changes with each verification request since the initial value of checksum variable and output of the PRG will be different for each challenge sent by the verifier. Hence, the final checksum returned by the device will be a function of the verifier's challenge.

To prove to the verifier that the ICE verification function is actually computing the checksum over itself, we need to detect the two copy attacks mentioned in Section 3.2. To prove to the verifier, that the ICE verification function is executing from the correct locations in memory, the ICE verification function includes the value of the program counter (PC) into the checksum. To prove to the verifier that the checksum is computed over the correct locations in memory, the ICE verification function includes the data pointer, that is used to read the memory, into the checksum. Hence, when the checksum returned by the device to the verifier is correct, the verifier is assured that the program counter, data pointer and the contents of the region of memory over which the checksum was computed, all had the correct values.

If an attacker tries to launch either of the copy attacks mentioned in Section 3.2, the attacker will have to incorporate additional instructions into the ICE verification function to simulate the correct values for the PC and the data pointer. These additional instructions will slowdown the computation of the ICE checksum.

The ICE verification function uses an alternate sequence of additions and XOR operations to compute the checksum, thereby making the checksum computation non-associative. An alternate sequence of additions and XOR operations is non-associative because $a \oplus b + c$ is equivalent to $(a \oplus b) + c$, but not $a \oplus (b + c)$.

In order to make the checksum function non-parallelizable, we use the two preceding checksum values to compute the current checksum value. Also, the PRG generates its current output based on its last output.

Figure 4 shows the pseudocode of the ICE verification function. The ICE verification function iteratively computes a 128-bit checksum of the contents of memory. The pseudocode is presented in a non-optimized form for readability. It takes in a parameter y which is the number of iterations the ICE verification function should perform when computing the checksum. The 128-bit checksum is represented as an array of eight 16-bit values. The ICE verification function updated one 16-bit element of the checksum array in each iteration of its loop. To update a checksum element, the ICE verification function loads a word from memory, transforms the word that is loaded and adds the transformed value to the checksum element. The checksum element is then rotated left by one bit.

The random challenge sent by the verifier is 144 bits long. Of this, 128 bits are used to initialize the checksum array and 16 bits are used as the seed for the T function.

We use a 16-bit T function [17] as the PRG. T functions have the property that the i^{th} output only depends on outputs $1 \ldots i$. The particular T function we use in the pseudocode is $x \leftarrow x + (x^2 \vee 5)$. In practice, we should use a family of T functions because a T function starts repeating itself after it has generated all elements in its range. Another option for a PRG would be the RC4 stream cipher. However, T functions are very efficient, and their code can be easily showed to be non-optimizable.

To ensure that the intruder cannot modify a single byte, the checksum function needs to examine the entire memory content under verification. Previously, researchers propose to traverse the memory in pseudo-random order [16, 29]. This approach is undesirable, however, because it requires $O(n \log(n))$ memory reads to achieve high probability that each memory location was accessed at least once, where n is memory size. The ICE verification function makes multiple linear passes over memory, thus requiring only $O(n)$ accesses to touch every memory location with a probability of one. As the pseudocode shows, the data pointer is incremented during each iteration of the loop and then checked for bounds before each memory read.

Figure 4. ICE Pseudocode

```
//Input: y number of iterations of the verification procedure
//Output: Checksum C
//Variables: [code_start, code_end] - bounds of memory address under verification
//           daddr - address of current memory access
//           b - content of daddr
//           x - value of T function
//           l - counter of iterations
daddr ← code_start
for l = y to 0 do
   //T function updates x
   x ← x + (x^2 ∨ 5)
   //Read from memory address a
   b ← mem[daddr++]
   if daddr > code_end then
      daddr ← code_start
   end if
   //Calculate checksum. Let C be the checksum vector and j be the current index.
   C_j ← C_j + PC ⊕ (b ⊕ PC + l ⊕ C_{j−2}) ⊕ (x ⊕ daddr + C_{j−1}) + PC
   C_j ← rotate left(C_j)
   //update checksum index
   j ← (j + 1) mod 8
end for
```

3.4 Implementation of ICE

3.4.1 Sensor Node Architecture

We implemented ICE the Telos sensor nodes, the most recent platform of the Berkeley mote family. The Telos motes the use the MSP430 microcontroller from Texas Instruments. The MSP430 is a 16-bit von-Neumann architecture with 60K of Flash memory, and 2K of RAM. The microcontroller has a 8MHz CPU that features the RISC architecture and has 16 16-bit registers.

The ICE verification function uses all 16 CPU registers. Thus, the attacker does have any more free registers for any modifications it makes. For an architecure that has more registers, we can deny the availability of reigsters to the attacker by storing the checksum in registers and extending the size of the checksum until no free registers remain.

The MSP430 CPU has the following features. Operation with immediate operands take more CPU cycles than register-to-register operations. In general, this property holds for most CPU architectures. The program counter (PC) is a regular register. Hence, we can easily incorporate the PC value into the checksum. The CPU also has a hardware multiplier. The presence of the multiplier considerably speeds up the computation of the T function. However, the presence of a hardware multiplier is not absolutely necessary for the ICE verification function. In the absence of a hardware multiplier, the multiply operation in the T function can be simuluated or the T function can be replaced by RC4, which does not require any multiply operations.

Assembly Instruction	**Explanation**
//T function updates x	
mov r15, &MPY	load x into first operand of hardware multiplier
mov r15, &OP2	load x into second operand of hardware multiplier
bis #0x05, &RESLO	OR 5 into output of hardware multiplier, which holds x^2
add &RESLO, r15	$x \leftarrow x + (x^2 \vee 5)$
//reads memory at address	daddr, *and calculates checksum* (C_j at register 6)
mov r14+, r13	$r13 \leftarrow mem[daddr$++$]$
xor r0, r13	$r13 \leftarrow r13 \oplus PC$
add r12, r13	$r13 \leftarrow r13 + loopIndex$
xor r4, r13	$r13 \leftarrow r13 \oplus C_{j-2}$
add r0, r6	$C_j \leftarrow C_j + PC$
xor r13, r6	$C_j \leftarrow C_j \oplus r13$
mov r15, r13	$r13 \leftarrow x$(from T function)
xor r14, r13	$r13 \leftarrow r13 \oplus daddr$
add r5, r13	$r13 \leftarrow r13 + C_{j-1}$
xor r13, r6	$C_j \leftarrow C_j \oplus r13$
add r0, r6	$C_j \leftarrow C_j + PC$
rla r6	$C_j \leftarrow$ rotate left$[C_j]$
adc r6	

Figure 5. ICE Assembly code

3.4.2 Assembly Code

Figure 5 shows the main loop the ICE verification function written in the assembly language of MSP-430. As can be seen, all variables used in the checksum computation are maintained in the CPU registers. The code is manually optimized to ensure that the attacker cannot find a more optimized implementation. The main loop consists of just 17 assembly instructions and takes 30 machine cycles. We will show that the best attack code would achieve a 3 cycle overhead in each iteration of the main loop, which represents a 10% overhead.

As part of the assembly code optimization, we unrolled the loop 8 times. This allows us to the keep the checksum array in the CPU registers and also to eliminate the checksum index variable. In the unoptimized code, bounds checking is performed on the data address at every memory access. After unrolling the loop, an obvious optimization would be to perform bounds checking at the very end of the unrolled loop instead of at every instance, thus saving cycles. If we do so, the data pointer might go out of bounds by at most 7 memory locations. To ensure the checksum function still operates correctly, we pad the end of the self verification code with known values (e.g., NOPs no-operation instructions) for up to 7 memory locations. Thus, if our memory reads are going out of bounds, we would still only be accessing known values.

Based on the assembler code, we will now show that the attacker incurs a time overhead of least 3 CPU cycles when it carries out any of attacks mentioned in Section 3.2. To carry out the memory copy attacks, the attacker has to forge the values of either the PC or the data pointer. The attacker does not have any free registers. Thus, the fastest way to forge the data pointer is save the correct value of the data pointer before the memory access, replace it with a value of the attacker's choosing and to restore the correct the value of the data pointer before it is incorporated into the checksum. This incurs an overhead of 4 CPU cycles per iteration on the MSP430.

To forge the PC, the attacker can replace the value of the PC by immediate since the each sampled value of a PC is a constant. However, on the MSP430 architecture (and most RISC architectures), such an operation using an immediate operand required 1 more CPU cycle compared to a register-to-register operation. Since we use the PC 3 times in each iteration, the attacker would incur a 3 CPU cycle penalty.

All other attacks that involve making changes to the ICE verification function code directly will involve the data substitution attack. The data substituion attacks requires that the attacker to insert at least one extra `if` statement into every iteration of the ICE verification function. An `if` statement translates into a compare and a branch in assembly. On the MSP430 a compare and a branch together take 3 CPU cycles.

Typically, the ICE verification function would verify itself as well as a few other functions that will execute immediately following it. After computing and returning the checksum, the ICE verification function would jump to one of these verified functions. From the assembler code, it is clear that the ICE verification function does not contain any contiguous memory region that has the same value. However, we cannot make the same claim about the other functions that ICE verifies. If these functions have a contiguous region all of which has the same value, like a buffer of zeros for example, the attacker can take advantage of this situation by having a malicious verification function that does not perform memory reads when it iterates through this memory region. In this way, the attacker would save some CPU cycles that could be used to carry out other attacks. To prevent this attack, we encrypt all memory content under verification except the code of the ICE verification function itself.

3.5 Results

We implemented two versions of the ICE verification function on the Berkeley Telos motes: a legitimate version and a malicious version that assumes that the attacker has a 3 CPU cycle overhead per iteration of the ICE verification function. This translates into a 10% runtime overhead. The MSP-430 microcontroller has an emulator board and a real-time C-SPY debugger that can monitor the status, register file, and memory content of the device. We profiled both executions and Figure 6 shows the runtime overhead. A detectable time difference is required in order for the ICE protocol to identify malicious nodes. As our results show, we can achieve an arbitrarily large time difference by varying the number of memory accesses.

Since the running time of the ICE verification function increases linearly with the number of iterations, we wish to minimize this number, and yet induce a time overhead to the attacker that is detectable by the verifier. In practice, the verifier should choose the number of iterations to ensure that the attacker's overhead is greater than network latency. As a corollary, we need a strict upper bound on network latency.

4 Protocols for Intrusion Detection and Repair

We start this section by describing how ICE can be used to construct the FIRE protocols, i.e., protocols for intrusion detection, code updates and cryptographic key updates in sensor networks. This is followed

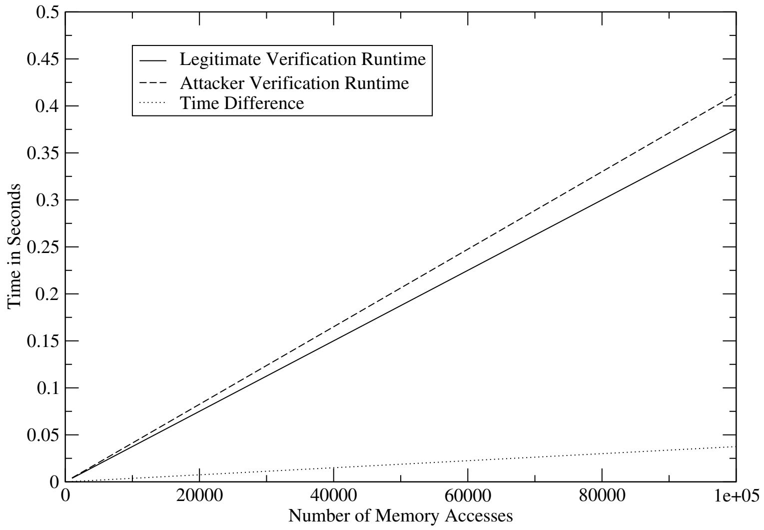

Figure 6. Runtime Overhead of Attacker

by a high-level description of the protocols. Appendix A, gives a more detailed description of the protocols. In Section 4.4, we discuss some points to be considered when using the FIRE protocols for building systems.

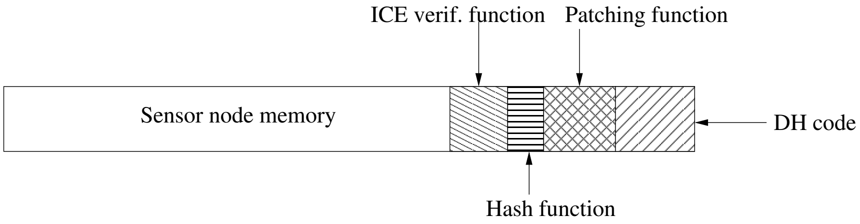

Figure 7. Memory layout of a sensor node. The ICE verification function is used to verify the contents of memory containing itself, and a hash function. The hash function computes a hash of the rest of the sensor node memory. The patch function is used for code updates and the Diffie-Hellman code is used for cryptographic key updates.

4.1 Extending ICE

The FIRE protocols use ICE as the primitive. The ICE verification function is a self-verifying function. When the ICE checksum returned by a device is correct the verifier is assured that the code that is expected to have executed on the device did in fact execute on the device.

We can ask the ICE verification function to produce a checksum of memory regions of any size. As long as the memory region being verified includes the portion of memory where the ICE verification function resides, a correct ICE checksum is a guarantee that the memory region over which the checksum was computed has the expected contents. When designing the FIRE protocols using ICE, we make the ICE verification function check a region of memory that contains the code for the FIRE protocols and the code for the ICE verification function. After computing the checksum, the ICE verification function jumps to the code of one of the FIRE protocols. In the context of sensor networks, the base station functions as the verifier and the sensor node is the device being verified. The memory of every sensor node has code for the ICE verification function and the FIRE protocols.

The time taken by a sensor node to compute the ICE checksum has to be measured accurately to verify the correctness of the ICE checksum. In multi-hop sensor networks, the network latency between the base station and the sensor node can considerably vary the time between sending a ICE challenge and receiving a response. To minimize the variance, the base station can ask a node that is the neighbor of the node computing the checksum to measure the ICE checksum computation time. However, the node that is asked to time the checksum computation of a another node has to be trusted. The base station establishes the required trust relationship by using an expanding ring method. The base station first verifies nodes that are one network hop away from it. In this case, the base station can directly time the ICE checksum computation. The nodes that are one network hop away are then asked by the base station to measure the time taken by their neighbors to compute the checksum. In this manner, the ICE verification spreads out from the base station like an expanding ring.

Even a one-hop network latency is not deterministic in a wireless environment where multiple sensor nodes contend for the radio channel. To make the one hop network latency deterministic, the sensor node computing the ICE checksum is given exclusive access to the radio channel. Now, the one hop network latency can be predetermined by the base station. The sensor node computing the ICE checksum can be be asked by the base station to do a sufficient number of iterations of the ICE verification function loop, so that the expected time overhead for an attacker's ICE checksum computation is much greater than the predetermined one hop network communication latency.

A malicious sensor node can forward the ICE challenge to a proxy node that has a copy of the correct memory contents and can compute the ICE checksum faster than the sensor node. The time saved by a faster computation of the ICE checksum can be used for communicating the ICE challenge from the sensor node to the proxy and communicating the ICE checksum from the proxy to the sensor node. This way the malicious sensor node can fake the ICE checksum and the forgery will go undetected by the base station.

To compute the checksum faster than a sensor node, the proxy has to be a computing device with greater computing and storage resources than a sensor node. For example, the proxy device can be a PC on the Internet, but not another sensor node. All sensor nodes having identical memory layouts take the same amount of time to compute the ICE checksum.

The base station detects proxy attacks by delaying all packets between the sensor network and the outside world by a few seconds when a sensor node is computing the ICE checksum. The base station is the gateway between the sensor network and the outside world. Any packets sent and received between a malicious sensor node and a proxy that is outside the sensor network will have to pass through the base station. If a sensor node tries to use a proxy node to help compute the ICE checksum, the delay introduced into the communication by the base station will ensure that the sensor node cannot return the ICE checksum to the base station within the expected amount of time.

4.2 Protocols for Intrusion Detection and Code Updates

We define intrusion as the process of a remote attacker compromising a node using software vulnerabilities. The purpose our intrusion detection and repair protocol is to provide a method for the base

$B \to A:$	$\langle$ICE Challenge$\rangle$
$B:$	$T_1 =$ Current time
$A:$	Compute ICE checksum over memory region containing ICE verification function and the hash function
$A \to B:$	$\langle$ICE checksum$\rangle$
$B:$	$T_2 =$ Current time Verify $(T_2 - T_1) \leq$ Allowed time to compute ICE checksum Verify ICE checksum from sensor node using checksum computed by self
$A \to B:$	$\langle$Hash of rest of memory$\rangle$
$B:$	Use hash of sensor node memory contents to determine if rest of sensor node memory is correct Prepare code patches for sensor node
$B \to A:$	$\langle$Code patches$\rangle$
$A:$	Apply patches

Figure 8. Protocol for detecting intrusion and sending code updates between the base station B and a sensor node A.

station to detect whether a node has been compromised. If any intrusion is detected, either the sensor node is blacklisted by the base station or the node is repaired. To repair a node, the base station sends an update to the sensor node to patch the software vulnerability. The base station is guaranteed that the sensor node applies the code updates sent by the base station and is repaired fully.

The intrusion detection mechanism does not make any false negative claims. That is, if a node is flagged by the mechanism as being uncompromised, the node is actually uncompromised. If a node is tagged as being compromised, then either the node is actually compromised or is experiencing a DOS attack. For example, a malicious node could jam an uncompromised node computing the ICE checksum. Then the node computing the ICE checksum will not be able to return the ICE checksum to the base station within the expected amount of time and will appear to have been compromised.

The ICE verification function computes a ICE checksum over the region of memory that contains the ICE verification function and a hash function. After finishing the ICE checksum computation, the ICE verification function jumps to the hash function. The hash function computes a hash over the rest of the memory. Figure 8 shows a simplified version of the intrusion detection and code update protocol. Appendix A gives the full protocol.

When the ICE checksum is correct, the hash function can be trusted to compute a correct hash of the sensor node's memory. In this case, the base station can compare the hash of a sensor node's memory with the correct hash to determine if there have been changes to the memory contents of the sensor node. The base station can also pinpoint exactly which locations in the memory of a sensor node have been changed by asking the sensor node to compute hashes of different regions of its memory. Once the changed locations in the memory of a sensor node have been identified, the base station can send memory updates for exactly those memory locations that have been modified on the sensor node. So, the amount of data sent from the base station to the sensor node will be minimized. Even though computing the extra hashes over memory take up energy, energy required for communication is at least an order of magnitude larger than energy used for computation. Hence, overall less energy will be utilized.

$B \to A:$	$\langle$ICE Challenge, DH half-key $g^y mod\ p\rangle$
$B:$	$T_1 =$ Current time
$A:$	Compute ICE checksum over memory region containing ICE verification function, hash function and node ID $C_1 =$ ICE checksum $r \xleftarrow{R} \{0,1\}^{128}$ Generate one-way hash chain, $d_2 = F(C_1) \oplus r, d_1 = F(d_2), d_0 = F(d_1)$
$A \to B:$	$\langle d_0, MAC_{C_1}(d_0)\rangle$
$B:$	$T_2 =$ Current time Verify $(T_2 - T_1) \leq$ Allowed time to compute ICE checksum Compute MAC of d_0 using ICE checksum computed by self If MAC of d_0 computed by self equals MAC of d_0 sent by sensor node, then node's ICE checksum is correct
$A:$	Compute hash of rest of memory H_{mem}
$A \to B:$	$\langle MAC_{d_1}(H_{mem})\rangle$
$A:$	Generate DH half-key $g^x mod\ p$
$A \to B:$	$\langle d_1, g^x mod\ p, MAC_{d_2}(g^x mod\ p)\rangle$
$B:$	Verify $d_0 = F(d_1)$ Compute $MAC_{d_1}(H_{mem})$ using d_1 and H_{mem} computed by self Verify MAC of H_{mem} returned by A
$A \to B:$	$\langle r\rangle$
$B:$	Compute $d_2 = F(C_1) \oplus r$ using r and ICE checksum computed by self Verify $d_1 = F(d_2)$ Verify MAC of $g^x mod\ p$ using d_2 Compute $(g^x mod\ p)^y mod\ p$
$A:$	Compute $(g^y mod\ p)^x mod\ p$

Figure 9. Protocol for symmetric key establishment between the base station B and a sensor node A. F is a cryptographic hash function based on the RC5 block cipher.The protocol uses a CBC-MAC derived from RC5.

If the ICE checksum returned by the sensor node is incorrect, then the memory region containing the ICE verification function, the hash function and the function to apply code updates has been modified. In this case the base station has no guarantee of what is actually executing on the sensor node. Thus, the base station blacklists the sensor node.

4.3 Cryptographic Key Update Protocol

Once a sensor node has been repaired by undoing changes the attacker made to its memory contents and the software vulnerability removed using a code update, a new cryptographic key needs to be established between the sensor node and the base station. Even if a sensor node looks uncompromised, a new cryptographic key needs to be established since the attacker could have compromised the node, read out its cryptographic key, and then undone all changes made to the memory.

Our cryptographic key update protocol does not rely on the presence of any shared secrets between the base station and the sensor node. We assume that the attacker knows the entire memory contents of the sensor node. The cryptographic key update protocol establishes a symmetric key between the base station and a sensor node, preventing man-in-the-middle and eavesdropping attacks.

At first glance, it may appear impossible to rule out man-in-the-middle and eavesdropping attacks without leveraging a shared secret key. However, the properties we rely on here are that, one, the attacker is remote and has a longer delay for messages and, two, all sensor nodes in the network have equal computational capabilities. Using the ICE approach, the base station sends a challenge that only a node with the correct memory contents can correctly solve. We assume that each sensor node has a few bytes of Read-Only Memory (ROM) containing its node id. The sensor node uses the challenge sent by the base station to compute a checksum over the memory region containing the ICE verification function, a hash function and its node id. The sensor node with the correct node id and memory layout will be able to generate the ICE checksum faster than all other nodes in sensor network. We leverage this asymmetry in time of computing the ICE checksum to establish a symmetric key between the base station and the sensor node.

A symmetric key is established between the base station and sensor node using the Diffie-Hellman (DH) key exchange protocol. In order to prevent man-in-the-middle attacks, the sensor node and the base station need to authenticate the DH protocol messages. We assume that all sensor nodes have the base station's public key in their Read-Only Memory (ROM). Hence, the sensor node can authenticate the base station's DH half key. A simple way to complete the DH key exchange is for the sensor node to generate and send a DH half key to the base station immediately after computing the ICE checksum. The sensor node also sends a MAC of its DH half key to the base station. The MAC is generated using the ICE checksum as the key. If the time taken by the sensor node to compute the ICE checksum and generate its DH half-key is less than the time taken by the attacker to forge the ICE checksum, then, on the verifying the MAC, the base station is assured that the DH half key could have only come from the correct sensor node. This statement is true since no other sensor node can compute the ICE checksum as fast the correct sensor node.

However, computing DH half keys is too slow on sensor nodes. An attacker can pre-compute a Diffie-Hellman half key before the ICE challenge arrives from the base station and then use the extra time to forge the ICE checksum. The attacker can then generate the correct MAC for its DH half-key.

Since generation of the Diffie-Hellman half key is too slow to perform right after computing the ICE checksum, we need a fast mechanism to set up an authenticated channel between the node and the base station. This authenticated channel can be used by the sensor node to send its Diffie-Hellman half key to the base station. Since one-way functions are efficient to compute, we use the Guy Fawkes protocol by Anderson et al. [2] to set up the authenticated channel. Both the base station and the sensor node compute a short one-way hash chain. We let the node create a fresh one-way hash chain, containing three elements, right after the checksum computation. The node generates the initial element of its hash chain as a function of the ICE checksum and a randomly chosen value. This ensures that an attacker cannot precompute the hash chain to save some time for forging the ICE checksum. In addition, since the node also uses a random value to generate the its hash chain, no other node can generate the node's hash chain even when after forging the ICE checksum sometime in the future.

The node uses the ICE checksum to authenticate its one-way chain to the base station. Then, the node computes a fresh Diffie-Hellman half key, and authenticates it through the Guy Fawkes protocol. Thus, we achieve secure key establishment without shared secrets, robust to man-in-the-middle attacks and eavesdropping by colluding malicious nodes. Figure 9 shows a simplified version of our key update protocol where we do not show details of how the sensor node authenticates packets from the base station. Appendix A gives the full protocol.

4.4 Considerations for System Design

Selection of cryptographic primitives Because of our application onto sensor nodes with limited computation power and resources, implementation of the cryptographic primitives posed a major challenge. To save program memory, we reuse one block cipher to implement all cryptographic primitives. We suggest using RC5 [28] as the block cipher because of its small code size and efficiency. In prior work in sensor network security [26], Perrig et al. stated that an optimized RC5 algorithm can compute an 8 byte block encryption in 120 cycles. Thus, on this architecture, one execution of RC5 merely requires 0.015 ms.

A CBC-MAC operation can be implemented by using RC5 as the block cipher. The hash function can also be constructed with RC5 as follows: $h(x) = RC5(x) \oplus x$, using a standard value as the key.

Diffie-Hellman parameters Because of stringent resource constraint on sensor nodes, most work on sensor network security only operates with symmetric cryptographic protocols. Generally, it is considered impractical to perform expensive asymmetric cryptographic operations on sensor nodes because they do not have enough computation power or memory size. In our work, by carefully picking the parameters, it is possible to run asymmetric algorithms on the Berkeley Telos motes.

By selecting the bare minimum needed to perform Diffie-Hellman, we used a subset of the TinyPK package from BBN [4]. The Diffie-Hellman key exchange is an asymmetric cryptographic protocol that is based on the operation $g^x \mod p$. The security of Diffie-Hellman is based on the length of secret x and a public p. When applied to sensor nodes, a 14 byte p and 64 byte x is sufficient, since it would yield a subgroup discrete logarithm key size of 112 bits. According to Lenstra et al. [18], these parameters would be deemed as secure in the year 1990 using state of the art technology at that time. Since we are dealing with low cost, mass quantity sensor nodes, 1990 levels of security is sufficient. Of course, the attacker can break our system using powerful Gigahertz machines for each sensor node, but this would be a very unlikely scenario because the attacker would incur a high cost.

Since g is relatively unimportant in the security of the protocol, we set g to be 2 in order to speed up computation. Using these parameters, the Telos mote were able to perform $g^x \mod p$ in 13.8 seconds. Since the Diffie-Hellman calculation is not timed as part of the ICE loop, a runtime of 13.8 s is acceptable.

5 Related Work

In this section, we review related work in code attestation, intrusion detection in wireless networks, code updates in wireless networks, and key distribution in wireless networks.

Hardware based attestation is promoted by the Trusted Computing group (TCG) [32]. Several chip manufacturers sell Trusted Platform Modules (TPMs), which implement the TCG standard. TCG and Microsoft's NGSCB have been proposed as memory-content attestation techniques that use secure hardware in form of a TPM chip to provide attestation [25, 32]. Due to cost and power constraints, sensor nodes are unlikely to have secure hardware. Also, TCG and NGSCB provide load-time attestation i.e.

they can only guarantee what was initially loaded into memory initially was correct. ICE requires runtime attestation to know what the current contents of memory are.

In the software-based attestation space, Kennel and Jamieson propose the first system [16], however Shankar, Chew, and Tygar have identified weaknesses in their work [30]. Seshadri et al. propose SWATT, which is a software-based memory content attestation mechanism [29]. SWATT needs to check the entire memory of the node to ensure that an attacker cannot hide malicious code anywhere in memory. Checking the entire memory is time consuming on nodes with large memory sizes. Further, SWATT does a pseudorandom access pattern over memory. This requires SWATT to perform O($n * \ln n$) accesses to memory, where n is the size of memory in bytes, to ensure that every memory location is accessed with high probability. This approach is impractical for large memories. Our ICE technique only checks a portion of memory instead of the whole memory, relieving this drawback. Our attestation performs a linear pass over memory. Thus, all memory locations are accessed with a probability of one.

Zhang and Lee [33] describe the issues of intrusion detection systems (IDS) in ad hoc wireless networks. They describe an architecture for an IDS for wireless networks. Marti et al. [24] propose an intrusion detection system specifically for the DSR routing protocol, their Watchdog and Pathrater attempt to find nodes that do not correctly forward traffic by identifying the attacking nodes and avoiding them in the routes used. Buchegger and LeBoudec [6] propose CONFIDANT, a system consisting of a monitor, a trust monitor, a reputation system, and a path manager. Lee et al. studied intrusion detection in wireless networks in more detail [1, 13, 33, 34]. All these approaches rely on passive network monitoring to detect malicious activity. These techniques all have false positives and false negatives. The approaches we describe in this paper take an active approach, by checking the memory of a node our technique is not susceptible to false negatives, however, an attacker who interferes with the verification may delay, corrupt, or jam the response message and thus cause false positives. In any case, this work is the first work that we are aware of that proposes an intrusion detection system for sensor networks.

In the area of sensor network software updates, all related research projects we are aware of do not consider security, but are mainly concerned with efficiency and reliability [14, 19, 20, 31]. They all assume a trustworthy environment.

Many researchers have considered key establishment protocols, however, all these efforts assume the presence of secret information to prevent man-in-the-middle attacks [3, 7, 9, 10, 21, 22, 26, 27, 35]. The key establishment protocol we present in this paper is the first sensor network routing protocol that prevents man-in-the-middle attacks without assuming the presence of authentic or secret information, or a trusted side-channel to establish authentic information.

6 Conclusion

We present a new architecture to secure sensor networks, which enables secure detection and recovery from sensor node compromise. Our approach is to design an intrusion detection system that is free of any false negatives, and that can identify compromised nodes. In addition, we design two mechanisms to recover compromised nodes, which to the best of our knowledge are the first protocols to deal with such issues. Our first mechanism can securely update the code of a sensor node, offering a strong guarantee that the node has been correctly patched. Our second mechanism sets up new cryptographic keys, even though an attacker may know all memory contents of the node, and can eavesdrop on and inject arbitrary messages in the network. All our mechanisms are based on ICE (Indisputable Code Execution), which freezes the memory contents to verify the correctness of the code currently executing on the node. Through our implementation in off-the shelf sensor nodes we demonstrate that our techniques are practical on current sensor nodes, without requiring specialized hardware. We are excited about other applications that our techniques may enable, which we will explore in our future work.

References

[1] Yi an Huang, Wei Fan, Wenke Lee, and Philip S. Yu. Cross-feature analysis for detecting ad-hoc routing anomalies. In *Proceedings of The 23rd International Conference on Distributed Computing Systems (ICDCS)*, May 2003.

[2] R. Anderson, F. Bergadano, B. Crispo, J. Lee, C. Manifavas, and R. Needham. A new family of authentication protocols. *ACM Operating Systems Review*, 32(4):9–20, October 1998.

[3] Ross Anderson, Haowen Chan, and Adrian Perrig. Key infection: Smart trust for smart dust. In *Proceedings of IEEE International Conference on Network Protocols (ICNP 2004)*, October 2004.

[4] BBN. Tinypk. Private communications, 2003.

[5] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *Advances in Cryptology - Crypto '96*, pages 1–15, Berlin, 1996. Springer-Verlag. Lecture Notes in Computer Science Volume 1109.

[6] Sonja Buchegger and Jean-Yves Le Boudec. Performance analysis of the confidant protocol (cooperation of nodes - fairness in dynamic ad-hoc networks). In *ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 2002)*, Lausanne, June 2002.

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

[8] John R. Douceur. The Sybil attack. In *First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002.

[9] W. Du, J. Deng, Y. Han, and P. Varshney. A pairwise key pre-distribution scheme for wireless sensor networks. In *Proceedings of the Tenth ACM Conference on Computer and Communications Security (CCS 2003)*, pages 42–51, October 2003.

[10] L. Eschenauer and V. Gligor. A key-management scheme for distributed sensor networks. In *Proceedings of the 9th ACM Conference on Computer and Communication Security*, pages 41–47, November 2002.

[11] Free Software Foundation. superopt - finds the shortest instruction sequence for a given function. `http://www.gnu.org/directory/devel/compilers/superopt.html`.

[12] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[13] Yian Huang and Wenke Lee. Attack analysis and detection for ad hoc routing protocols. In *Proceedings of The 7th International Symposium on Recent Advances in Intrusion Detection (RAID 2004)*, September 2004.

[14] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, November 2004.

[15] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: a goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 304–314, 2002.

[16] Rick Kennell and Leah H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 11th USENIX Security Symposium*. USENIX, August 2003.

[17] Alexander Klimov and Adi Shamir. New cryptographic primitives based on multiword t-functions. In *Fast Software Encryption, 11th International Workshop*, February 2004.

[18] Arjen Lenstra and Eric Verheul. Selecting cryptographic key sizes. In *Journal of Cryptology: The Journal of the International Association for Cryptologic Research*, 1999.

[19] Philip Levis, Sam Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric Brewer, and David Culler. The emergence of networking abstractions and techniques in TinyOS. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, March 2004.

[20] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, March 2004.

[21] D. Liu and P. Ning. Establishing pairwise keys in distributed sensor networks. In *Proceedings of the Tenth ACM Conference on Computer and Communications Security (CCS 2003)*, pages 52–61, October 2003.

[22] D. Liu and P. Ning. Location-based pairwise key establishments for static sensor networks. In *ACM Workshop on Security in Ad Hoc and Sensor Networks (SASN '03)*, October 2003.

[23] D. Malan, M. Welsh, and M. Smith. A public-key infrastructure for key distribution in TinyOS based on elliptic curve cryptography. In *Proceedings of the First IEEE International Conference on Sensor and Ad hoc Communications and Networks (SECON 2004)*, October 2004.

[24] Sergio Marti, T.J. Giuli, Kevin Lai, and Mary Baker. Mitigating routing misbehaviour in mobile ad hoc networks. In *Proceedings of the sixth annual International Conference on Mobile Computing and Networking*, pages 255–265, Boston MA, USA, August 2000.

[25] Next-Generation Secure Computing Base (NGSCB). `http://www.microsoft.com/resources/ngscb/default.mspx`, 2003.

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

[27] Roberto Di Pietro, Luigi V. Mancini, and Alessandro Mei. Random key assignment for secure wireless sensor networks. In *ACM Workshop on Security of Ad Hoc and Sensor Networks (SASN 2003)*, November 2003.

[28] Ron Rivest. The RC5 encryption algorithm. In Ross Anderson, editor, *Proceedings of the 1st International Workshop on Fast Software Encryption*, volume 809 of *Lecture Notes in Computer Science*, pages 86–96. Springer-Verlag, Berlin Germany, 1995.

[29] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Swatt: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[30] Umesh Shankar, Monica Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *Proceedings of USENIX Security Symposium*, pages 89–101, August 2004.

[31] Thanos Stathopoulos, John Heidemann, and Deborah Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing, November 2003.

[32] Trusted Computing Group (TCG). `https://www.trustedcomputinggroup.org/`, 2003.

[33] Yongguang Zhang and Wenke Lee. Intrusion detection in wireless ad-hoc networks. In *Proceedings of International Conference on Mobile Computing and Networking (MobiCom 2000)*, August 2000.

[34] Yongguang Zhang, Wenke Lee, and Yian Huang. Intrusion detection techniques for mobile wireless networks. *ACM/Kluwer Wireless Networks Journal (ACM WINET)*, 9(5), September 2003.

[35] S. Zhu, S. Setia, and S. Jajodia. LEAP: Efficient security mechanisms for large-scale distributed sensor networks. In *Proceedings of the Tenth ACM Conference on Computer and Communications Security (CCS 2003)*, pages 62–72, October 2003.

A Appendix

Figure 10 shows the the protocol that is used by the base station (B) detecting intrusion in a sensor node and sending code updates to a sensor node (A). The base station and the sensor node are one network hop away from each other. The protocol is for a node which has not been compromised but has a software vulnerability that needs to be patched. The base station has a private key, denoted in the protocol by K_B^{-1}.

Figure 11 shows the detailed protocol for symmetric key establishment between the base station, B, and a sensor node, A. The base station has a private key, denoted in the protocol by K_B^{-1}.

B :	$h_4 \overset{R}{\leftarrow} \{0,1\}^{144}$
	Generate one-way hash chain $h_3 = F(h_4), h_2 = F(h_3), h_1 = F(h_2), h_0 = F(h_1)$
$B \rightarrow A$:	$\langle h_0, \{h_0\}_{K_B^{-1}} \rangle$
A :	Verify signature on h_0 using base station's public key from ROM
B :	Wait for 15 secs to allow node to verify signature on h_0
$B \rightarrow A$:	$\langle h_1 \rangle$
B :	$T_1 =$ Current time
A :	Verify $h_0 = F(h_1)$
	Compute ICE checksum over memory region containing ICE verification function, hash function and node ID using h_1 as key
	$C_1 =$ ICE checksum
	$r \overset{R}{\leftarrow} \{0,1\}^{128}$
	Generate one-way hash chain $d_2 = F(C_1) \oplus r, d_1 = F(d_2), d_0 = F(d_1)$
$A \rightarrow B$:	$\langle d_0, MAC_{C_1}(d_0) \rangle$
B :	$T_2 =$ Current time
	Verify $(T_2 - T_1) \leq$ Allowed time to compute ICE checksum
	Verify MAC of d_0 by recomputing MAC using ICE checksum computed by self
	If MAC of d_0 computed by self equals MAC of d_0 sent by sensor node, then node's ICE checksum is correct
$B \rightarrow A$:	$\langle h_2 \rangle$
A :	Verify base station's acknowledgment (h_2), $h_1 = F(h_2)$
	Compute hash of rest of memory
	$H_{mem} =$ Hash of memory
$A \rightarrow B$:	$\langle MAC_{d_1}(H_{mem}) \rangle$
$B \rightarrow A$:	$\langle h_3 \rangle$
A :	Verify base station's acknowledgment (h_3), $h_2 = F(h_3)$
$A \rightarrow B$:	$\langle d_1 \rangle$
B :	Verify, $d_0 = F(d_1)$
	Compute $MAC_{d_1}(H_{mem})$ using d_1 and H_{mem} computed by self
	Verify MAC of H_{mem} returned by A
$B \rightarrow A$:	$\langle codepatch, MAC_{h_4}(codepatch) \rangle$
$A \rightarrow B$:	$\langle r \rangle$
B :	Compute $d_2 = F(C_1) \oplus r$ using r and ICE checksum computed by self
	Verify, $d_1 = F(d_2)$
$B \rightarrow A$:	$\langle h_4 \rangle$
A :	Verify, $h_3 = F(h_4)$
	Compute and verify $MAC_{h_4}(codepatch)$ using h_4
	Apply patch

Figure 10. Protocol used by the base station, B, detect intrusion and to send a code update to patch a software vulnerability in an uncompromised sensor node, A. F is a cryptographic hash function based on the RC5 block cipher. The protocol uses a CBC-MAC derived from RC5.

$B:$ $h_4 \xleftarrow{R} \{0,1\}^{144}$
Generate one-way hash chain $h_3 = F(h_4), h_2 = F(h_3), h_1 = F(h_2), h_0 = F(h_1)$

$B \to A:$ $\langle h_0, \{h_0\}_{K_B^{-1}} \rangle$

$A:$ Verify signature on h_0 using base station's public key from ROM

$B:$ $y \xleftarrow{R} \{0,1\}^{112}$
Wait for 15 secs to allow node to verify signature on h_0

$B \to A:$ $\langle h_1, g^y \ mod \ p, MAC_{h_2}(g^y \ mod \ p) \rangle$

$B:$ $T_1 =$ Current time

$A:$ Verify, $h_0 = F(h_1)$
Compute ICE checksum over memory region containing ICE verification function,
hash function and node ID using h_1 as key
$C_1 =$ ICE checksum
$r \xleftarrow{R} \{0,1\}^{128}$
Generate one-way hash chain, $d_2 = F(C_1) \oplus r, d_1 = F(d_2), d_0 = F(d_1)$

$A \to B:$ $\langle d_0, MAC_{C_1}(d_0) \rangle$

$B:$ $T_2 =$ Current time
Verify $(T_2 - T_1) \leq$ Allowed time to compute ICE checksum
Verify MAC of d_0 by recomputing MAC using ICE checksum computed by self
If MAC of d_0 computed by self equals MAC of d_0 sent by sensor node, then ICE checksum computed by node is correct

$B \to A:$ $\langle h_2 \rangle$

$A:$ Verify base station's acknowledgment (h_2), $h_1 = F(h_2)$
Verify MAC of $g^y \ mod \ p$ using h_2
Compute hash of rest of memory
$H_{mem} =$ Hash of memory

$A \to B:$ $\langle MAC_{d_1}(H_{mem}) \rangle$

$B \to A:$ $\langle h_3 \rangle$

$A:$ Verify base station's acknowledgment (h_3), $h_2 = F(h_3)$
$x \xleftarrow{R} \{0,1\}^{112}$

$A \to B:$ $\langle d_1, g^x \ mod \ p, MAC_{d_2}(g^x \ mod \ p) \rangle$

$B:$ Verify, $d_0 = F(d_1)$
Compute $MAC_{d_1}(H_{mem})$ using d_1 and H_{mem} computed by self
Verify MAC of H_{mem} returned by A

$B \to A:$ $\langle h_4 \rangle$

$A \to B:$ $\langle r \rangle$

$B:$ Compute $d_2 = F(C_1) \oplus r$ using r and ICE checksum computed by self
Verify, $d_1 = F(d_2)$
Verify MAC of $g^x \ mod \ p$ using d_2
Compute $(g^x \ mod \ p)^y \ mod \ p$

$A:$ Verify base station's acknowledgment (h_4), $h_3 = F(h_4)$
Compute $(g^y \ mod \ p)^x \ mod \ p$

Figure 11. Protocol for symmetric key establishment between the base station B and a sensor node A. F is a cryptographic hash function based on the RC5 block cipher. The protocol uses a CBC-MAC derived from RC5.