# SCUBA: Secure Code Update By Attestation in Sensor Networks*


Arvind Seshadri† CMU/CyLab
Mark Luk‡ CMU/CyLab
Adrian Perrig§ CMU/CyLab
Leendert van Doorn¶ IBM
Pradeep Khosla‖ CMU/CyLab



## ABSTRACT

This paper presents SCUBA (Secure Code Update By Attestation), for detecting and recovering compromised nodes in sensor networks. The SCUBA protocol enables the design of a sensor network that can detect compromised nodes without false negatives, and either repair them through code updates, or revoke the compromised nodes. The SCUBA protocol represents a promising approach for designing secure sensor networks by proposing a first approach for automatic recovery of compromised sensor nodes. The SCUBA protocol is based on ICE (Indisputable Code Execution), a primitive we introduce to dynamically establish a trusted code base on a remote, untrusted sensor node.



**Categories and Subject Descriptors:** Software, Operating Systems, Security and Protection.

**General Terms:** Security.

**Keywords:** Externally-verifiable Code Execution, Software-based Attestation, Secure Code Update, Self-checksumming Code.


## 1. INTRODUCTION

Sensor networks are expected to be deployed in the near future in many safety-critical applications such as critical infrastructure protection and surveillance, alarm systems, home and office automation, inventory control systems, and many medical applications such as patient monitoring. Hence, if an attacker injects malicious code into the sensor nodes, it can compromise the safety and privacy of users.

We consider the setting of a sensor network where an attacker has compromised sensor nodes by injecting malicious code into their memory. The base station wants to verify the code memory contents of the nodes, and either repair the nodes by undoing any changes made by the attacker or blacklist the nodes which cannot be repaired.

This paper presents the SCUBA (Secure Code Update By Attestation) protocol, which enables the base station to perform code updates to sensor nodes. The protocol assumes that the update has to be performed in the presence of malicious code that might try to interfere with the update. For example, if the base station sends a code patch, malicious code on the node may fake the installation of the patch. To make the problem tractable, we assume that the attacker's hardware devices are not present in the sensor network for the duration of the repair process. Even with this assumption, our protocol represents a significant benefit since circumventing it requires the attacker's hardware to be always present in the sensor network. This significantly increases the attacker's exposure compared to the situation today where the attacker need only be physically present intermittently in order to compromise sensor nodes.

To the best of our knowledge, we present the first protocol for secure code updates to recover from node compromise in sensor networks. The base station, which sends the code update, obtains a firm guarantee either that the code update was correctly installed (thereby undoing all changes made by the attacker) or that the malicious code running on the sensor node is preventing the application of the code update. In the latter case, the base station can blacklist the sensor node. We assume commodity sensor nodes (i.e., no special hardware required). The SCUBA protocol can recover nodes even if an attacker compromises an arbitrary number of nodes, uploads arbitrary code into the compromised nodes, and if compromised nodes arbitrarily collude.

In order to securely perform the code update, the base station needs to obtain a guarantee that no malicious code will interfere with the execution of the SCUBA protocol on the sensor node. The Pioneer primitive can be used to obtain this guarantee [20]. Pioneer, which is implemented for the x86 architecture, allows an external verifier to obtain the guarantee of *untampered code execution* on an untrusted computing platform. That is, the external verifier obtains an assurance that no malicious code present on the untrusted computing platform can interfere with the execution of some arbitrary executable that the verifier wants to invoke on the untrusted computing platform. We port the Pioneer primitive to the TI MSP430, the CPU used on the Telos rev.B sensor nodes [16]. We call our primitive for the MSP430 CPU ICE (Indisputable Code Execution) since it enables the base station (the external verifier) to obtain an *indisputable* guarantee that the SCUBA protocol executable on the sensor node (the untrusted computing platform) will execute untampered.

The base station commands the sensor node to invoke the ICE verification function, which sets up an execution environment for untampered execution of the SCUBA protocol executable. The ICE verification function is constructed so that any attempt by the attacker to fake the creation of the correct execution environment will be detected by the base station. When the base station does not detect any such attempt, it is assured that the SCUBA protocol code will execute on the node without interference from malicious code.


*This research was supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, and grant CNS-0347807 from the National Science Foundation, and by a gift 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.



†arvinds@cs.cmu.edu
‡mark.luk@gmail.com
§perrig@cmu.edu
¶T.J. Watson Research Center, leendert@us.ibm.com
‖pkk@ece.cmu.edu




*WiSe'06,* September 29, 2006, Los Angeles, California, USA.

The base station then sends a code update to the SCUBA protocol on the node, which installs the code update thereby removing the changes made by the attacker.

**Outline** In Section 2, we describe the problem definition, the sensor network architecture, and assumptions, and the attacker model. Section 3 discusses prior research in the areas of untampered code execution and software-based attestation. Section 4 describes the ICE primitive. In Section 5, we describe the SCUBA protocol. Section 6 discusses related work, and Section 7 concludes.

# 2. PROBLEM DEFINITION, ATTACKER MODEL, AND ASSUMPTIONS

In this section, we first define our problem in the context of sensor networks. We then discuss our attacker model. Finally, we state our assumptions about the sensor network, the base station and the sensor nodes.

## 2.1 Problem Definition

We consider the setting of a sensor network where an attacker has compromised sensor nodes by injecting malicious code into their memory. The base station wants to verify the code memory contents of the nodes, and either repair the nodes by undoing any changes made by the attacker or blacklist the nodes which cannot be repaired. The repair needs to be done in the presence of malicious code that may interfere with the repair.

## 2.2 Attacker Model

In this paper, we study an attacker who compromises sensor nodes by injection of malicious code. Malicious nodes controlled by the attacker can collude. We assume that the attacker's hardware devices are not present in the sensor network for the duration of the repair process. While this is a strong assumption, the SCUBA protocol represents a significant advance since circumventing it requires the attacker's hardware to be always present in the sensor network. This significantly increases the attacker's exposure compared to the situation today where the attacker need only be physically present intermittently in order to compromise sensor nodes. 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 devices.

## 2.3 Sensor Network Architecture

We consider a wireless sensor network consisting of one or multiple base stations and several sensor nodes. All sensor nodes have have the same make and model of CPU and identical memories. The sensor nodes communicate among themselves and the base station using a wireless network. Every sensor node and the base station has a unique identifier, hereafter referred to as *node ID* or *base station ID*. 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. Sensor nodes can send and receive packets to hosts on external networks only through the base station.

## 2.4 Assumptions

The base station knows the make and model of the CPU on the sensor nodes. 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 [15], 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 [18]. We assume that the base station is not compromised by the attacker. This assumption is commonly made in secure sensor networks, since compromise of the base station implies compromise of the entire network.

We assume that the base station and sensor nodes share a cryptographic key. However, when the attacker compromises a node, it could learn the node's key. Therefore, as part of undoing the changes made by the attacker, the base station also needs to establish a new key with the node. The issue is one of key establishment: how can we establish a key between the base station and a node without relying the pre-existence of shared secrets between them? The untampered code execution mechanism provided by ICE can be used to perform this task. We do not give the details here since it is out of the scope of this paper. A preliminary version of our protocol for key establishment based on ICE is available [2]. A forthcoming paper discusses a better version of the protocol.

We also 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, the attacker cannot modify the memory region containing the node ID even after it compromises the sensor node. The ICE verification function uses the sensor node's node ID as part of the input used to generate the checksum. By doing so, we leverage ICE to prevent impersonation attacks where an attacker changes the node ID of a node to impersonate another node, for example as in the Sybil attack [5]. We discuss our defense in detail in Section 4.3. 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 in compromised nodes.

The sensor network is assumed to provide a reliable transport layer protocol like PSFQ [23]. The SCUBA protocol requires that protocol messages be reliably delivered between participants.

# 3. BACKGROUND

In this section, we discuss prior research in the areas of software-based untampered code execution, and software-based attestation. We focus on software-based techniques because sensor nodes are unlikely to have hardware extensions for attestation or untampered code execution due to cost, size, and power concerns.

Pioneer is the first software-based technique that provides the guarantee of untampered code execution, the property that we need to perform secure code updates on compromised sensor nodes [20]. However, Pioneer relies on hardware-dependent techniques specific to the Pentium platform, and thus cannot be used on the CPU of sensor nodes. ICE is the only software-based primitive that we are aware of which achieves untampered code execution on simple CPU architectures.

An area that is closely related to untampered code execution is that of software-based attestation. Attestation enables a verifier to check the integrity of software present on a computing platform. However, attestation by itself cannot guarantee untampered code execution since the attacker can modify the code between the time the code is verified and the time the code is invoked for execution. This is referred to as a time-of-check-to-time-of-use (TOCTTOU) attack. Hence, none of the techniques we discuss below can be used to address the problem of secure code updates on compromised sensor nodes.

Spinellis proposed using self-verifying code as a means to verify software integrity [6]. The work uses the setting of an external verifier that wants to verify the integrity of some piece of code on a device. The external verifier asks the device the split the memory region being verified into two overlapping regions. The start and end addresses of these two regions differ from one verification request to another. The device generates a hash of each region separately and returns the hashes to the verifier. Since the memory regions that are hashed vary from verification to verification, the attacker cannot pre-compute or replay the hash values. The external verifier has a copy of the code being verified and can hence check the correctness of the hashes returned by the device. Spinellis also proposes using CPU state such as the behavior of caches or performance counters as inputs to the checksum to detect attempts by the attacker to modify the hash computation in order to forge the correct hash values.

Kennell and Jamieson propose a system, Genuinity, that uses ideas similar to those of Spinellis in the context of the x86 architecture [11]. The main difference between the two works is that Genuinity adds the use of time as a side channel to detect cheating by the attacker.

SWATT is a software-based memory attestation technique that allows an external verifier to perform an equality check on the memory contents of an embedded device [19]. SWATT is targeted towards simple CPUs and uses the idea to pseudorandom memory traversal and uses time as a side channel to detect attempts by the attacker to manipulate the verification process. The setting of ICE is similar to that of SWATT [19]. However, as we mentioned earlie, SWATT can only provide a guarantee of memory content integrity, unlike ICE which provides the stronger guarantee of untampered code execution. Also, unlike ICE, SWATT needs to check the entire memory contents of the device. Checking the entire memory contents of a device might be impractical since part of the device's memory might have contents that are unknown to the verifier (e.g., dynamic data or cryptographic secrets). The ability to check memory sizes smaller than the entire memory of the device also reduces the energy consumed by ICE to perform the verification of memory when compared to SWATT.

PIV is a technique to perform program integrity checks on sensor nodes [22]. The primary idea is that the verifier creates a new randomized hash function for each verification request and sends the hash function to the sensor node along with the verification request. However, PIV does not time the hash computation on the sensor node, and could be vulnerable to dynamic attacks where the attacker manipulates the execution of the hash function on the sensor node to generate the correct checksum despite having made modifications to the program running on the sensor node.

Shaneck et al. propose a technique for remote attestation of sensor nodes [17]. The verifier constructs a new attestation procedure for each verification request and sends the code to the sensor node being verified. The attestation procedure uses various code obfuscation techniques to make it hard for the attacker to perform static or dynamic analysis of the attestation procedure within the time alloted to the sensor node by the base station for computing the attestation response. However, the paper does not have any results to substantiate this claim. Further, the authors state that the base station should set the expected time for getting a response from a sensor node to the sum of the network round trip time, the time taken to compute the attestation procedure on the sensor node, and the expected delay in response due to factors like network delay. As long as the base station can determine an upper bound on the expected delay due to random factors like network jitter, SWATT can be used in the setting of the paper to perform attestation of sensor nodes. Also, unlike ICE, the authors do not consider impersonation attacks and they assume that the sensor network is a one-hop network.

# 4. ICE: INDISPUTABLE CODE EXECUTION

In this section, we first give an overview the ICE primitive and show how we use self-checksumming code to build the ICE verification function. We then discuss various attacks against the ICE verification function and show how we design the defences against them. Finally, we describe our implementation of the ICE verification function on the Telos rev.B motes.

## 4.1 Indisputable Code Execution Overview

We consider the model where the base station wants to invoke some executable on the sensor node. The sensor node could have malicious code which may attempt to interfere with the execution. The base station wants to obtain the guarantee that the execution of the executable on the sensor node has not been tampered with in any manner. We refer to this guarantee as *untampered code execution*.

The following three step process is one way to achieve untampered code execution. 1) Check the integrity of the executable. This ensures that the executable has not been modified before it is invoked for execution. 2) Set up an execution environment in which the execution of the executable is guaranteed to be atomic. That is, once it starts executing, no other code on the sensor node is allowed to execute until the executable exits. We refer to such an execution environment as an *untampered execution environment*. 3) Invoke the executable to execute in the untampered execution environment. It is important that the three steps outlined above be executed atomically. Otherwise, the attacker can carry out a time-of-check-to-time-of-use (TOCTTOU) attack. In this attack, the attacker modifies the executable after the integrity check but before the executable is invoked for execution.

We construct a challenge-response protocol, called ICE (Indisputable Code Execution) between the base station and sensor node to enable the base station to obtain the guarantee of untampered code execution. The sensor node has a function called the ICE verification function which carries out the three step process described in the previous paragraph to ensure that the execution of an arbitrary executable on the sensor node will not be tampered by any malicious code that exists on the sensor node. However, since the ICE verification function is also a piece of software, malicious code could tamper with the execution of the ICE verification function. How do we ensure that the ICE verification function itself executes untampered?

The ICE verification function is a *self-checksumming* code. We define self-checksumming 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. This provides us with a means to verify the integrity of the ICE verification function executable. However, malicious code may attempt to tamper with the execution of the ICE verification function, even when the executable image of the ICE verification function is correct. To prevent this attack, the ICE verification function sets up an untampered execution environment for its execution before it starts to compute the checksum. This consists of setting up CPU state to ensure atomic execution of the ICE verification function. An example of setting CPU state to ensure atomic execution is disabling interrupts. The ICE verification function takes the CPU state used to set up the untampered execution environment as input to generate the checksum. The ICE verification function is constructed so that if any part of the relevant CPU state is incorrect then either the checksum will be incorrect or the checksum computation would be slower. Thereby, a correct checksum that is generated within the expected amount of time guarantees that the ICE verification function will execute untampered.

Figure 1 shows an overview of ICE. The base station sends a "check integrity and execute" request to the sensor node. The ICE verification function on the sensor node computes a checksum as a function of the memory region containing its own instruction sequence, the instruction sequence of the executable, and the CPU state that needs to be set up to create an untampered execution environment. The ICE verification function returns the checksum to the base station and invokes the executable. Since the executable is directly invoked by the ICE verification function, the executable "inherits" the untampered execution environment of the ICE verification function and hence executes untampered by malicious code.

The base station has a copy of the ICE verification function code and the executable. It also knows the correct value of the CPU state necessary to create the untampered execution environment on the sensor node. Hence, the base station can verify if the checksum re-

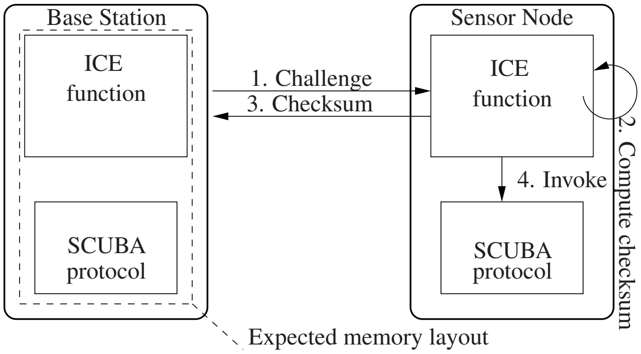


**Figure 1: Overview of ICE. The numbers represent the temporal ordering of events.**

turned by the sensor node is correct. Since the base station knows the CPU make and model on the sensor node, it knows the expected time to compute the checksum on the sensor node. If the base station receives the correct checksum from the sensor node within the expected time, the base station obtains the guarantee of untampered execution of the executable on the sensor node.

## 4.2 Attacks Against Self-Checksumming Code

In order to fake the untampered execution of the executable, the attacker has to fake the correct checksum within the expected time even though it modifies either the memory region containing the executable and the ICE verification function, or sets up the CPU state corresponding to the untampered execution environment incorrectly. Such attacks can be classified into three types. In the first type of attack, an attacker attempts to forge the checksum locally on the sensor node. In the second type of attacks, the adversary attempts to speed up the checksum computation, either locally or by use helper devices. Finally, in the third type, malicious nodes in a sensor network may attempt to use impersonation attacks. For example, a malicious node may attempt to have an unmodified sensor node compute the response to the base station's challenge by forwarding the base station's challenge to the unmodified node. Having given an overview of ICE in Section 4.1, we now describe how we design our defenses against these attacks.

### *4.2.1 Checksum Forgery Attacks*

The attacker needs to forge the value despite modifying the memory contents being checksummed or having an incorrect CPU state corresponding to the untampered execution environment. The different attacks in this category are as follows:

**Pre-computation and replay attacks.** An attacker may attempt to compute the checksum over the memory region containing the ICE verification function and the target executable, before making changes to the memory region. 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 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 dictionary attacks when the attacker stores previously observed challenge-checksum pairs.

**Data substitution attacks.** An attacker may attempt to change some locations of the memory region containing the ICE verification function and the executable 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. This attack can be detected by having the ICE verification function access memory in pseudorandom pattern. Thereby the attacker cannot predict in advance which memory accesses by the ICE verification function will read the memory locations modified by the attacker. The attacker is then forced to check every memory access by the ICE verification function. The extra checks slow down the attacker's checksum computation. This approach is similar to the one in our earlier work on SWATT [19].

We use the result of the Coupon Collector's Problem to guarantee that the checksum code will read every memory location of the ICE verification function and the executable with high probability, despite the pseudo-random memory access pattern. If the size of the ICE verification function and the executable is $n$ words, the result of the Coupon Collector's Problem states: if $X$ is the number of memory reads required to read each of the $n$ words at least once, then $Pr[X > cn\ln n] \leq n^{-c+1}$. Thus, after $O(n\ln n)$ memory reads, each memory location is accessed at least once with high probability.

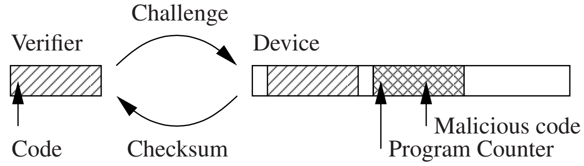


**Figure 2: In this attack, the correct code resides at a different memory location, and the attacker executes malicious code at the correct memory location, computing the memory checksum over the correct code.**

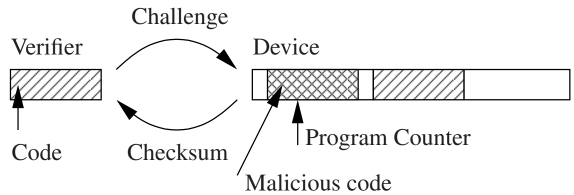


**Figure 3: In this attack, the correct code resides at the correct memory location, but the attacker executes malicious code at a different memory location, computing the memory checksum over the correct code.**

**Memory copy attacks.** Since we only want to verify a small part of the memory 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-checksumming 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.

To detect the two memory copy attacks, we use the program counter and the data pointer as part of the data used to compute the checksum. Either of the copy attacks requires the attacker 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 slow down checksum computation.

**Forging CPU state inputs.** We mentioned earlier that setting up the untampered execution environment is equivalent to ensuring atomic execution of the executable. ICE guarantees atomic execution by setting up the corresponding CPU state and incorporating them into the checksum. An attacker may attempt to forge the creation of the untampered execution environment by incorrectly setting up the CPU state and then forging the correct CPU state during checksum computation. We now describe this potential attack in detail and argue why such an attack would fail.

The CPU state required for atomic execution varies depending on the CPU architecture. In the following discussion, we will focus on architecturally simple CPUs that are commonly employed on sensor nodes. Such processors lack advanced architectural features such as support for virtual memory, memory protections, or caches. On such CPUs, atomicity of execution can be achieved by disabling interrupts during execution. Since the CPUs under consideration do not support exceptions, disabling interrupts ensures that no other code can execute.

Interrupts are of two kinds: maskable and non-maskable. Setting the `interrupt-disable` bit to the appropriate state disables maskable interrupts. Non-maskable interrupts, however, cannot be disabled. Therefore, in the presence of non-maskable interrupts, we cannot guarantee execution atomicity of the executable. Instead we relax our requirement of atomicity to state that only code that has been verified by the ICE verification function be allowed to execute during the execution of the executable. With this relaxed requirement, we include a default handler for non-maskable interrupts in the ICE verification function. This is a dummy handler that simply executes an `interrupt-return` instruction to return control to whatever code was executing when the non-maskable interrupt occurred. The ICE verification function modifies the CPU's interrupt vector table so that the interrupt vectors for all the non-maskable interrupts point to the dummy interrupt handler within the ICE verification function. With this change, any non-maskable interrupt that occurs during the execution of the executable will cause control to be unconditionally returned to ICE.

By including the `interrupt-disable` bit and the CPU's interrupt vector table in the checksum, we ensure that the checksum is correct only if these inputs are correct. If the attacker sets up (one of) these CPU states incorrectly, the attacker will have to forge the correct value during checksum computation thereby leading to a time overhead.

#### 4.2.2 Attacks to speed up Checksum Computation

The attacker may attempt to speed up checksum computation in two ways: use a helper device or do it on the sensor node. The first attack mentioned below attempts to speed the checksum computation on the sensor node. The second and the third attack try to use helper devices.

**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 [11,19], we need to show that the code cannot be further optimized. We can use automated tools to either exhaustively find the most efficient implementation [8], or to use theorem proving techniques to show that a given code fragment is optimal [10]. 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 may attempt to 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 make the verification function non-parallelizable to force sequential execution.

**Proxy attack.** A malicious sensor node may attempt to forward the ICE challenge to a proxy node with greater computing resources. The proxy node has a copy of the correct memory contents of the sensor node. Having greater computing resources, the proxy node 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 forge the ICE checksum and the forgery will go undetected by the base station. We call this attack the proxy attack.

We assume in this paper that all sensor nodes in the network have identical computing resources and the attacker is not physically present during verification. Hence, the proxy will have to be a node outside the network (e.g., a PC on the Internet). Any packets sent by the sensor nodes to devices outside the network will have to pass through the base station since the sensor nodes only have short-range wireless links.

The base station prevents proxy attacks by blocking all network packets from outside the sensor network during the process of verification. Hence, any sensor node that tries to use an external proxy will be unable to receive the computed checksum from the proxy.

#### 4.2.3 Impersonation Attacks

A malicious sensor node may attempt to have a legitimate sensor node impersonate it in the ICE protocol by forwarding protocol messages from the base station to the legitimate sensor node. Also, a malicious sensor node may attempt to assume multiple identities (Sybil attack).

To prevent such attacks, the base station needs an assurance that the ICE response it receives comes from the sensor node the base station is trying to verify. We achieve this property through a few bytes of Read-only Memory (ROM) on every node. The ROM stores the node ID of the node. The node ID is unique to each node and since it is in a ROM the attacker cannot modify it, even when the attacker compromises the node. The ICE verification function uses the ROM-based node ID as an input to the checksum computation. By doing so, the attacker is forced to forge the node ID if it wants to do an impersonation attack. To perform this forgery, the attacker has to use a data substitution attack to insert a conditional check. This check would divert the read to the ROM to other memory location where the attacker has stored the node ID of the node the attacker is trying to impersonate. Thereby, the attacker's checksum computation is slowed down. Hence, the base station can detect impersonation attacks through the slowdown in checksum computation.

### 4.3 Design of the ICE Verification Function

We now discuss how we design the ICE verification function to implement the defenses we mentioned in the previous section.

**Keyed checksum.** The checksum computed by the device should be a function of the challenge sent by the verifier to prevent precomputation and replay attacks. We could use a cryptographic message authentication code (MAC), like HMAC [3] to generate the checksum. However, MAC functions have much stronger properties than we require. MACs are designed to resist a MAC forgery attack. In this attack, an attacker observes the MAC values for a number of different inputs, all of which are computed using the same MAC 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. Hence, the input used to compute the checksum changes

//Input: $y$ number of iterations of the verification procedure
//Output: Checksum $C$
//Variables:
// $[code\_start, code\_end]$: verified memory area
// $daddr$: address of current memory access
// $b$: content at $daddr$
// $x$: value of T function
// $l$: loop counter
// $SR$: status (flags) register
**for** $l = y$ to 0 **do**
  //T function updates $x$ where $0 < x < 2^16$
  $x \leftarrow x + (x^2 \vee 5) \mod 2^16$
  //Compute random memory address using $x$
  $daddr = ((daddr \oplus x) \wedge MASK) + code\_start$
  //Calculate checksum. $j$: current index into checksum vector.
  $C_j \leftarrow C_j + PC \oplus mem[daddr] + l \oplus C_{j-1} + x \oplus daddr +$
    $C_{j-2} \oplus SR$
  $C_j \leftarrow$ rotate left$(C_j)$
  //update checksum index
  $j \leftarrow (j+1) \mod 10$
**end for**

**Figure 4: ICE Pseudocode.**

with each verification request and so the final checksum returned by the device will be a function of the verifier's challenge.

**Pseudo-random number generator.** We use a 16-bit T-function as the PRG [12]. A T-function is a bijection from n-bit words to n-bit words. Certain T-functions also have the property that their single cycle length is equal to $2^n$, where $n$ is the size of the input to the T-function in bits. The particular T-function we use in the ICE verification function, $x \leftarrow x + (x^2 \vee 5)$ where $\vee$ is the bitwise or operator, has this property.

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 shown to be optimal.

**Preventing memory copy attacks.** Figure 3 and Figure 2 illustrates the two memory copy attacks. In the first one, the attacker executes malicious code at the correct memory location, while computing the checksum over the correct code residing elsewhere. This could be implemented either by faking the data pointer used to read memory, or displacing all memory reads by an offset. In the second attack, the correct code resides at the correct memory location, but the attacker executes malicious code elsewhere. This would require the attacker to forge the value of the program counter when used as an input to the checksum. By incorporating the program counter and data pointer into the checksum, such potential attacks would result in extra computation, which would slow down checksum computation.

**Strongly-ordered checksum function.** The ICE verification function uses an alternate sequence of additions and XOR operations to compute the checksum. This sequence of operations has the property that the final value of checksum will be different with high probability if the sequence of operations is altered in any way. A strongly-ordered checksum function prevents the attacker from computing the checksum out-of-order or in parallel. It also prevents the attacker from removing operations from the checksum function or replacing them with other operations in an attempt to speed up checksum computation.

**Non-parallelizable.** 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.

## 4.4 Implementing ICE

This section describes our implementation of the ICE verification function on the TI MSP430 micro-controller that is used on the Telos rev.B motes.

**Overview of the TI MSP430.** The MSP430 is a 16-bit RISC CPU that uses the von-Neumann architecture. It has 48KB of Flash memory, 10KB of RAM, and uses a 8MHz clock. There are 16 16-bit general purpose registers, `r0` through `r15`. Of these `r0` is the PC, `r1` is the stack pointer, `r2` is the status (flags) register, and `r3` is the constant generator. This leaves 12 registers available for general purpose use by programs. 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 simulated or the T-function can be replaced by RC4, which does not require any multiply operations. The `interrupt-disable` bit on the MSP430 is part of the status register, `r2`. Also, the interrupt vector table resides in the top 32 bytes of memory.

**Pseudocode.** Figure 4 shows the pseudocode of the ICE verification function. The ICE verification function iteratively computes a 160-bit checksum. The pseudocode is presented in a non-optimized form to improve readability. It takes in a parameter $y$ which is the number of iterations the ICE verification function should perform when computing the checksum.

The ICE verification function uses `r13` to hold the loop counter, which is initialized to $y$, `r14` to hold the data pointer used to read memory, and `r15` to hold the current value of the T-function output. This leaves 10 registers available, including the stack pointer. The ICE verification function can use the stack pointer as a general purpose register since it does not use the stack. To ensure that the attacker does not have any free registers available the ICE verification function uses all 10 registers to hold the checksum, resulting in a 160-bit checksum value.

To efficiently compute the checksum that is held in 10 general purpose registers, we unroll the ICE verification function loop 10 times. Each unrolled code block updates one of the registers holding the checksum. After unrolling the loop, an obvious optimization is to decrement the loop counter by 10 at the very end of the unrolled loop instead of performing it in every code block.

Each code block includes the memory word read, the PC, the data pointer, the output of T-function, the loop counter, and the two previously computed checksum values in the checksum. The code blocks also include the status register in the checksum to check the status of the `interrupt-disable` flag. The memory region containing the interrupt vector table is also used as an input to compute the checksum. Pseudo-random memory traversal is implemented by xoring the current value of the T-function with the data pointer and appropriately masking the result to ensure that the data pointer falls within the memory region of interest.

The challenge sent by the verifier is 128 bits long. This challenge is used to initialize 8 of the 10 16-bit registers that hold the checksum. The ninth and the tenth register are initialized to a value computed by xoring the first four and last four 16-bit words of the 128-bit challenge. The 16-bit seed for the T-function is generated by xoring together the eight 16-bit words that make up the 128-bit challenge.

**Assembly code.** Figure 5 shows one code block of the unrolled loop of the ICE verification function written in the assembly language of MSP430. The code is manually optimized to ensure that the attacker cannot find a more optimized implementation. The code block consists of 17 assembly instructions and takes 32 CPU cycles. Incorporating the PC into the checksum is made simple by the fact that the PC can be treated as a general purpose register. This property holds for most RISC CPUs.

| 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 | $x \leftarrow x + (x^2 \vee 5) \mod 2^n$ |
| *//modifies address* daddr, *based on* x *from T function* | |
| xor r14, r6 | $daddr \leftarrow daddr \oplus x$ |
| and #0x1FF, r6 | mask last few bits of $daddr$ |
| add #ICE_LOOP, r6 | $daddr \leftarrow daddr + \text{ICE\_LOOP}$ |
| *//reads memory at address* daddr, *and calculates checksum* | |
| add r0, r6 | $C_j \leftarrow C_j + PC$ |
| xor @r14, r6 | $C_j \leftarrow C_j \oplus mem[daddr]$ |
| add r13, r6 | $C_j \leftarrow C_j + loopIndex$ |
| xor r5, r6 | $C_j \leftarrow C_j \oplus C_j - 1$ |
| add r15, r6 | $C_j \leftarrow C_j + x$ (from T function) |
| xor r14, r6 | $C_j \leftarrow C_j \oplus daddr$ |
| add r4, r6 | $C_j \leftarrow C_j + C_{j-2}$ |
| xor SR, r6 | $C_j \leftarrow C_j \oplus$ status register |
| rla r6 | $C_j \leftarrow$ rotate left$[C_j]$ |
| adc r6 | |

**Figure 5: ICE Assembly code**

## 4.5 Results

In this section, we show the attacker's overhead for the attacks described in Section 4.2. The design of the checksum function only allows the attacker to use the data substitution attack or the memory copy attacks to forge the checksum. If the attacker uses any of the other attacks, the final checksum will be wrong. We developed the fastest implementation of both attacks and timed their execution.

Since the ICE verification function uses all 16 CPU registers, the attacker does not have any more free registers. Therefore, the attacker can only use immediate or memory operands in its code. With this constraint, the two fastest implementations of the memory copy attack are as follows. In the first method, the attacker replaces all accesses to the PC with immediates. This allows the attacker to keep an unmodified code image at the expected location while executing a modified ICE function elsewhere. This modified ICE function would forge the PC by using an immediate instead of the actual PC. In the second method, the attacker displaces all memory reads by a fixed constant. This allows the attacker to execute a modified ICE function at the expected location, while redirecting all memory reads to an unmodified code image placed at a constant offset.

Both memory copy attacks incur a one-cycle overhead per code block. On the MSP430 architecture, a register-to-register operation takes one cycle, while an immediate-to-register operation takes two cycles. Thus, in the first attack, the attacker incurs a one-cycle overhead every time it forges the PC value with an immediate. Another feature of the MSP430 architecture is that memory reads with displacement addressing require three cycles, while direct memory reads require two cycles. By replacing direct memory access with displaced reads, the second memory copy attack also incurs a one-cycle penalty per code block. Since each code block consists of 32 CPU cycles, a memory copy attack would have an overhead of 1/32, of 3.1%. We implemented this attack and timed the execution, both on the node (local timing), and by the base station (one hop RTT).

In the data substitution attack, the attacker needs to check every memory read by inserting an `if` statement. This translates into a compare instruction and a jump instruction. The compare instruction uses an immediate or memory operand and hence requires at least 2 cycles to execute. The jump instruction also takes 2 cycles to execute. Thus, this attack suffers an overhead of 4/32, or 12.5%.

**Selecting number of loop iterations.** The base station and the node communicate over a network link. The attacker may attempt to reduce the latency of network communication to gain time to forge the checksum. The theoretically best attacker has zero network latency. Therefore, if we estimate a worst-case bound on the network latency, then we can always detect the existence of an attacker if its time overhead to forge the checksum is greater than the worst-case network latency.

Estimating the worst-case network latency in a multi-hop sensor network is not easy. In Section 5.2 we develop a technique that allows the checksum computation time of a node to be always observed by a node that is one-hop away. Therefore, we only need to estimate the worst-case one-hop network latency. However, even the one-hop latency is not deterministic in a wireless environment where multiple nodes contend for the radio channel. To make the one hop network latency deterministic, the node computing the ICE checksum is given exclusive access to the radio channel. Now, the worst-case one hop network latency can be predetermined.

The attacker's overhead to forge the checksum is directly proportional to the number of iterations of ICE verification function. So, in our experiments we choose the number of iterations so that the attacker's overhead for its fastest attack is still higher than the worst-case one-hop network latency of the sensor network. The time allowed to compute the checksum is set to be the expected computation time plus the worst-case one-hop network latency. If the perceived execution time by the verifier is higher than the threshold, the device executing the ICE verification function is assumed to be compromised. A false negative is the case of a device verifying correctly even though it is compromised, while a false positive occurs when a legitimate device is classified to be malicious. In this setting, false negatives are impossible since a malicious ICE verification function would not be able to forge the checksum within the expected checksum computing time. However, if the actual latency of a legitimate packet was higher than the worst-case one-hop latency we allow for, a false positive would occur.

The Telos motes have a radio interface that communicates at 250 Kbps. The lowest overhead attack for the attacker is the memory copy attack, which has a 3% attacker overhead. The worst-case one-hop communication latency between two nodes was measured by having them continually exchange packets over a period of time and monitoring for the maximum latency experienced, which was 51ms. Based on these data, we choose the number of iterations to be 40,000.

**Experimental setup.** There are two Telos motes within direct communication range, one acting as the base station, and one acting as the sensor node running the ICE verification function. We implemented the ICE protocol between these two nodes on TinyOS. We implemented two versions of the ICE verification function: a legitimate function and a malicious function employing the memory copy attack. Execution timing measurements were taken by both nodes, and the experiment was repeated for the legitimate and the malicious verification functions. Hence, Figure 6 reports four sets of timing measurements. Timing measurements taken by the base station is the sum of the execution time of the ICE verification function and the one-hop network RTT, while timing measurements by the node being verified only consist of the running time of the ICE verification function. The expected time to compute the checksum was set to be the execution time of the legitimate ICE verification function plus maximum one-hop network latency (51ms). As our results show, the base station that is one hop away from the node was always be able to observe the attacker's time overhead.

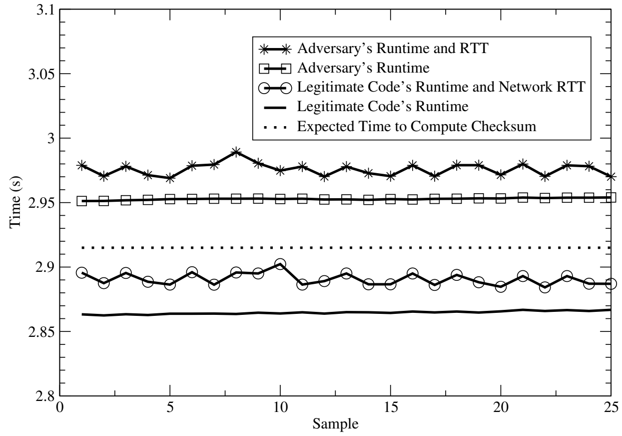


**Figure 6: Results of ICE and the fastest implementation of the memory copy attack, running 40,000 iterations.**

| | |
|---|---|
| $B \to A$ : | $\langle$ICE Challenge$\rangle$ |
| $B$ : | $T_1$ = Current time |
| $A$ : | Compute ICE checksum over memory region containing the ROM, the ICE verification function, and the SCUBA executable |
| $A \to B$ : | $\langle$ICE checksum$\rangle$ |
| $B$ : | $T_2$ = Current time<br>Verify $(T_2 - T_1) \leq$ Time allowed to compute ICE checksum<br>Verify ICE checksum from node by recomputing it |
| $A \to B$ : | $\langle$Hash of code memory$\rangle$ |
| $B$ : | Use hash from node to determine if node's code memory is modified<br>Prepare code patches for sensor node |
| $B \to A$ : | $\langle$Code patches$\rangle$ |
| $A$ : | Apply patches |

**Figure 7: SCUBA Protocol between the base station *B* and a sensor node *A*.**

# 5. PROTOCOL FOR SECURE CODE UPDATES

In this section we describe how ICE can be used to construct the SCUBA protocol. We start off with a high level description of the SCUBA protocol between the base station and a sensor node separated by one network hop. Appendix A gives a more detailed description of the protocol. Then, we discuss how the SCUBA protocol can be extended to work with sensor nodes that are multiple hops away from the base station. Finally, we discuss how the SCUBA protocol can be used to undo modifications made by the attacker to the data on the sensor nodes.

## 5.1 SCUBA Protocol

The purpose of the SCUBA protocol is to provide a method for the base station to repair a compromised sensor node through code updates. The compromised node could contain malicious code that will interfere with the code update process. The SCUBA protocol will either blacklist the node or repair it. To repair a node, the base station sends a code update to undo any changes made to software on the node by the attacker.

Figure 7 shows a simplified version of the SCUBA protocol. The base station invokes the ICE verification function on the sensor node by sending a challenge. The ICE verification function computes a checksum over the memory region containing itself, the SCUBA protocol executable, and the ROM containing the base station's public key and the sensor node's node ID. If the base station receives the correct checksum from the sensor node within the expected time, the base station obtains the guarantee that the SCUBA protocol executable on the sensor node will execute untampered. In this case the base station will repair the node's software via code updates. If the checksum received by the base station is incorrect, or it takes too long to arrive, the base station presumes that malicious code on the node is interfering with the code update process and blacklists the node.

After computing and returning the checksum, the ICE verification function invokes the hash function within the SCUBA protocol executable. The hash function sends the hash of the node's code memory to the base station. The base station compares the hash returned by the sensor node with the correct hash value of the code memory to determine if there have been changes to the code memory contents of the sensor node. The base station can also pinpoint exactly which locations in the memory of a sensor node have been modified by the attacker by asking the sensor node to compute hashes of different regions of its memory. Once the modified 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 by the attacker. Thereby the amount of data sent from the base station to the sensor node will be minimized.

The base station and the sensor node need to authenticate the protocol packets they receive. The SCUBA protocol uses the Guy-Fawkes protocol to set up a two-way authenticated channel between the base station and the node [1]. To use the Guy-Fawkes protocol, both the base station and node generate short hash chains. However, before the hash chains can be used, both the base station and the node need to authenticate the first element of each other's hash chain. The base station sends the first element of its hash chain along with the digital signature of the element generated using its private key to the node. The node uses the base station's public key in its ROM to verify the digital signature. In this way, the node can authenticate the first element of the base station's hash chain.

To enable the base station to authenticate the first element of the node's hash chain, we make use of the fact that only the node with the correct memory layout will be able to generate the ICE checksum within the expected time. Since the memory content that is checked includes the node's node ID (which is immutable) only the node being verified (one with the correct node ID) will be able to generate the checksum within the expected time. After it finishes computing the checksum, the node sends the first member of its hash chain and a MAC of this element to the base station. The MAC is computed using the ICE checksum as the key. The base station independently generates the ICE checksum and can verify the MAC sent by the sensor node. If the MAC of the hash chain member sent by the node verifies correctly at the base station and the element and the MAC are received within the expected time, the base station is guaranteed that the hash chain element came from the correct node.

Appendix A presents the full SCUBA protocol, which shows how a two-way authenticated channel is established between the base station and the node.

## 5.2 Expanding Ring Method

The protocol description above deals with a sensor node that is one hop away from the base station. For a node that is multiple hops away from the base station, the time between sending an ICE challenge and getting a response can vary considerably due to variations in network latency. Since the time taken by a node to compute the ICE checksum has to be measured accurately to verify the correctness of the ICE checksum, we need a way to minimize the network latency variance for nodes that are multiple hops away.

We propose the following expanding ring method to minimize the network latency variance. The intuition behind our method is that if the checksum computation time of a node is always measured by a neighboring node, then the network latency is always that of a single hop. To achieve this condition, 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.

To verify a node that is multiple hops away, the base station first selects a verified neighbor of the node. The base station then sends an ICE challenge to the neighbor. The challenge is encrypted using the key that the neighbor shares with the base station. Encrypting the challenge prevents a colluding malicious node along the path between the base station and a neighbor from receiving the challenge before the node being verified. The malicious node may attempt to use the time saved by receiving the challenge earlier to forge the ICE checksum. The neighbor decrypts the challenge, issues it to the node being verified, and times the checksum computation. Upon receiving the checksum, the neighbor constructs a result packet containing the checksum and the time taken to compute the checksum. The neighbor computes a MAC of the result packet computed using the key it shares with the base station. The neighbor then sends the result packet to the base station along with the MAC. The base station uses the MAC to verify that the result packet it receives is authentic. The base station then verifies the correctness of the checksum in the result packet and the time taken to compute the checksum.

There are two issues in the expanding ring method that need to be addressed. One, the key which the neighbor shares with the base station needs to changed as part of the verification of the neighbor since the attacker might have compromised the neighbor and read out its key. Two, the neighbor may get compromised after it is verified but before it is asked by the base station to verify other nodes.

The first issue is one of key establishment: how can we establish a key between the base station and a node without relying the preexistence of shared secrets between them? The untampered code execution mechanism provided by ICE can be used to perform this task. We do not give the details here since it is out of the scope of this paper. A preliminary version of our protocol for key establishment based on ICE is available [2]. A forthcoming paper discusses a better version of the protocol.

To prevent the neighbor from getting compromised in the time between its verification and the time it is asked to verify other nodes, the verified nodes do not exit the SCUBA protocol executable to resume normal processing until the entire network has been verified by the base station. Due to its small code size, the SCUBA executable can be subjected to formal verification and manual audit to verify that it does not contain known software vulnerabilities in its code. Since any verified node continues to execute the SCUBA executable till it is explicitly asked to exit by the base station, the base station is assured that none of the verified nodes will get compromised till the verification process is complete. At the end of the verification process, the base station broadcasts an exit message to all nodes. Upon receiving this message, the nodes exit the SCUBA executable and resume normal processing.

### 5.3 Repairing Data

So far, we have discussed how the SCUBA protocol can be used to undo changes made by the attacker to the code in a sensor node. However, it is possible that the attacker modifies data present on the node in addition to the code. Then to repair a node fully, the base station would need to undo changes to the data as well.

The SCUBA protocol can easily be extended to address this. Data can be classified as static data and dynamic data. Static data is data that never changes. The base station knows the correct contents of the memory region on the node containing static data. Therefore, the method used to undo changes to static data is identical to the method used for code.

Dynamic data is data that is generated and modified by programs running on the node. Therefore, the base station cannot know the correct contents of memory regions on the node that contain dynamic data. Therefore, the base station asks the SCUBA protocol executable on the sensor node to reset the node after updating the node's code and static data. This way, the node starts executing from a clean state where all dynamic data is cleared to zero. The only caveat to this approach is that software on the node might use both volatile and non-volatile memory to store dynamic data. Since non-volatile memory is not cleared on reset, the SCUBA protocol executable must clear all dynamic data in non-volatile memory before resetting the node.

## 6. RELATED WORK

In this section, we review research related to software updates in sensor networks. We already discussed research in the areas of untampered code execution and software-based attestation in Section 3.

In the area of sensor network software updates, extensive research had been conducted in the context of efficiency and reliability, but assumes a trustworthy environment [9,13,14,21]. Deng et al. attempted to secure code updates by using Merkle hash trees and hash chains to authenticate the code distribution [4]. By leveraging authenticated streams, Dutta et al. secured Deluge, the de facto TinyOS network programming system [7]. However, these protocols only solve one half of the secure code update problem: how can the receiver authenticate code updates from the base station. In this paper, we solve the other half: how can the sender verify that the receiver indeed applied the code update. Combining the SCUBA with above approaches provides a complete solution for securing code updates.

## 7. CONCLUSION

We present SCUBA, a protocol that enables secure detection and recovery from sensor node compromise. To the best of our knowledge, this is the first protocol to deal with recovering sensor nodes after compromise. Our code update protocol can securely update the code of a sensor node, offering a strong guarantee that the node has been correctly patched, or detect when the patch failed.

Our protocol is based on ICE (Indisputable Code Execution), a primitive that can guarantee untampered execution of code even on a compromised node. ICE is a novel primitive in sensor networks, and applying it to secure code updates is just one of its many applications. The utility of ICE is only limited by the program verified as the target executable, and we envision that ICE would be a useful tool to develop other secure sensor network protocols.

There are some research issues outstanding in our current work. The assumption that attacker's hardware devices are not present in the sensor network during the repair is a strong one and we are working towards providing similar properties for a relaxed attacker model. Also, the ICE verification function needs to be subjected to further cryptanalysis.

Our implementation in off-the shelf sensor nodes shows 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.

## 8. REFERENCES

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

[2] A.Seshadri, M.Luk, A.Perrig, L. van Doorn, and P.Khosla. Using FIRE and ICE for detecting and recovering compromised nodes in sensor networks. Technical Report CMU-CS-04-187, School of Computer Science, Carnegie Mellon University, December 2004.

[3] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Advances in Cryptology - Crypto*, pages 1–15, 1996.

[4] J. Deng, R. Han, and S. Mishra. Secure code distribution in dynamically programmable wireless sensor networks. In *Proceedings in International Conference on Information Processing in Sensor Networks (IPSN 2006)*, 2006.

[5] J. Douceur. The Sybil attack. In *Proceedings of Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002.

[6] D.Spinellis. Reflection as a mechanism for software integrity verification. *ACM Transactions on Information and System Security*, 3(1):51–62, February 2000.

[7] P. Dutta, J. Hui, D. Chu, and D. Culler. Securing the deluge network programming system. In *Proceedings in International Conference on Information Processing in Sensor Networks (IPSN)*, 2006.

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

[9] J. Hui and D. 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)*, November 2004.

[10] R. Joshi, G. Nelson, and K. Randall. Denali: a goal-directed superoptimizer. In *Proceedings of ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 304–314, 2002.

[11] R. Kennell and L. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of USENIX Security Symposium*, August 2003.

[12] A. Klimov and A. Shamir. New cryptographic primitives based on multiword t-functions. In *Fast Software Encryption*, February 2004.

[13] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in TinyOS. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[14] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

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

[16] Moteiv Corp. *Tmote Sky: Low Power Wireless Sensor Module*, June 2006.

[17] M.Shaneck, K.Mahadevan, V.Kher, and Y.Kim. Remote software-based attestation for wireless sensors. In *ESAS*, pages 27–41, 2005.

[18] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. D. Tygar. SPINS: Security protocols for sensor networks. In *Proceedings of Conference on Mobile Computing and Networks (MobiCom)*, July 2001.

[19] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[20] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, October 2005.

[21] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, UCLA-CENS, November 2003.

[22] T.Park and K.Shin. Soft tamper-proofing via program integrity verification in wireless sensor networks. *IEEE Transactions on Mobile Computing*, 4(3), May/June 2005.

[23] C. Y. Wan, A. T. Campbell, and L. Krishnamurthy. PSFQ: A reliable transport protocol for wireless sensor networks. In *Proceedings of ACM Workshop on Wireless Sensor Networks and Applications (WSNA)*, September 2002.


# APPENDIX

## A. SCUBA PROTOCOL

Figure 8 shows the protocol that is used by the base station $B$ for verifying code integrity and sending code updates to a node $A$. For simplicity, the base station and the sensor node are one network hop away from each other. The base station has a public/private key pair, denoted in the protocol by $K_B$ and $K_B^{-1}$, respectively.

The SCUBA protocol uses the Guy-Fawkes protocol [1] to establish a two-way authenticated communication channel between the base station $B$ and sensor node $A$. The node generates its hash chain as a function of the ICE checksum. This prevents a malicious node from saving time to forge the ICE checksum by pre-computing the hash chain. The expected memory region consists of the memory region containing the ICE verification function, the SCUBA protocol code, the first member of the base station's Guy Fawkes hash chain ($h_0$ in Figure 8), and the ROM.

| | |
|---|---|
| $B:$ | $h_4 \xleftarrow{R} \{0,1\}^{128}$<br>Generate one-way hash chain $h_3 = F(h_4)$,<br>$h_2 = F(h_3), h_1 = F(h_2), h_0 = F(h_1)$ |
| $B \rightarrow A:$ | $\langle h_0, \{h_0\}_{K_B^{-1}} \rangle$ |
| $A:$ | Verify signature on $h_0$ using base station's<br>public key from ROM |
| $B:$ | Wait for node to verify signature on $h_0$ |
| $B:$ | $T_1 =$ Current time |
| $B \rightarrow A:$ | $\langle h_1 \rangle$ |
| $A:$ | Verify $h_0 = F(h_1)$<br>Compute ICE checksum over expected memory<br>region using $h_1$ as challenge<br>$C =$ ICE checksum<br>$r \xleftarrow{R} \{0,1\}^{128}$<br>Generate one-way hash chain $d_2 = F(C\|\|r)$,<br>$d_1 = F(d_2), d_0 = F(d_1)$ |
| $A \rightarrow B:$ | $\langle d_0, MAC_C(d_0) \rangle$ |
| $B:$ | $T_2 =$ Current time<br>Verify $(T_2 - T_1) \leq$ Allowed time<br>Verify MAC of $d_0$ by recomputing ICE checksum.<br>Abort if incorrect. |
| $B \rightarrow A:$ | $\langle h_2 \rangle$ |
| $A:$ | Verify base station's acknowledgment<br>$h_0 = F(F(h_2))$<br>Compute hash of code memory<br>$H_{mem} =$ Hash of code memory |
| $A \rightarrow B:$ | $\langle H_{mem}, MAC_{d_1}(H_{mem}) \rangle$ |
| $B \rightarrow A:$ | $\langle h_3 \rangle$ |
| $A:$ | Verify base station's acknowledgment<br>$h_2 = F(h_3)$ |
| $A \rightarrow B:$ | $\langle d_1 \rangle$ |
| $B:$ | Verify authenticity of $d_1$<br>$d_0 = F(d_1)$<br>Verify MAC of $H_{mem}$ returned by A<br>If $H_{mem}$ sent by $A$ equals known good value then stop |
| $B \rightarrow A:$ | $\langle codepatch, MAC_{h_4}(codepatch) \rangle$ |
| $A \rightarrow B:$ | $\langle r \rangle$ |
| $B:$ | Compute $d_2 = F(C\|\|r)$<br>Verify, $d_1 = F(d_2)$ |
| $B \rightarrow A:$ | $\langle h_4 \rangle$ |
| $A:$ | Verify, $h_3 = F(h_4)$<br>Compute and verify $MAC_{h_4}(codepatch)$ using $h_4$<br>Apply patch |

**Figure 8: The full SCUBA protocol.** ***B*** **is the base station,** ***A*** **is a potentially compromised node. F is a cryptographic hash function.**