Remote Software-Based Attestation for Wireless Sensors

Mark Shaneck, Karthikeyan Mahadevan, Vishal Kher, and Yongdae Kim

Computer Science and Engineering,
University of Minnesota - Twin Cities

Abstract. Wireless sensor networks are envisioned to be deployed in mission-critical applications. Detecting a compromised sensor, whose memory contents have been tampered, is crucial in these settings, as the attacker can reprogram the sensor to act on his behalf. In the case of sensors, the task of verifying the integrity of memory contents is difficult as physical access to the sensors is often infeasible. In this paper, we propose a software-based approach to verify the integrity of the memory contents of the sensors over the network without requiring physical contact with the sensor. We describe the building blocks that can be used to build a program for attestation purposes, and build our attestation program based on these primitives. The success of our approach is not dependent on accurate measurements of the execution time of the attestation program. Further, we do not require any additional hardware support for performing remote attestation. Our attestation procedure is designed to detect even small memory changes and is designed to be resistant against modifications by the attacker.

1 Introduction

Recent technological advances in hardware and communications have helped to achieve significant strides in the area of wireless sensor networks. These networks can be used in several real-world applications, including various critical applications, such as military surveillance, infrastructure security monitoring and fault detection (e.g., Golden Gate Bridge monitoring [23]), or industrial waste monitoring.

When sensors are deployed for critical applications, securing these sensors is important. If a sensor is compromised, an attacker can reprogram the sensor to act on his/her behalf. For example, the attacker can cause the sensor to send incorrect information to hide some military activity or send false information about the location of certain troops. Therefore, it is important to verify that the static memory contents of the sensors have not been modified, that is, to *attest* the static memory contents (which includes programs, keys, and system configuration information) of the sensors. Typically, sensors are deployed in large numbers in environments that may not be safe or easily accessible to humans. Further, the deployment mechanisms (e.g., unmanned air planes) often make it infeasible to locate the position of each sensor individually. Therefore, we need attestation mechanisms that do not require physical contact with the sensors, but rather use the wireless communication network. In other words, we need mechanisms to perform *remote* attestation.

In this paper we present a software-based approach to remotely attest the static memory contents of the sensors without requiring any additional hardware on the sensors. As sensors are inherently designed to be light-weight and inexpensive, adding additional hardware on the sensors significantly increases the cost well as the size of the sensors; therefore, software-based approaches are always preferable and practical (they also work on legacy systems). In our approach, in order to attest the sensor, the attester sends an attestation routine to the sensor and waits for some time (expected response time) to get a response from the routine. Once the response time elapses, the attester will not accept any response sent by the sensors. The sensor executes the routine, which randomly reads the sensor's static memory contents and returns a checksum of the memory contents. The attester has an exact image of the memory contents of each sensor and can pre-compute the checksum by locally running the attestation routine on the memory image. After receiving the checksum from the routine, the attester can verify whether the received checksum matches the expected result. Every attestation routine is unique per sensor and randomized so that the attacker will be unable to predict (and pre-compute the checksum) the next routine from the previous routines.

Motivation. One way of performing remote software-based attestation is to include a small attestation routine in the sensor's kernel that performs a checksum on the memory contents of the sensor. To prevent replay attacks, for every new attestation request, the attester sends a random key to the sensor and the routine on the sensor pseudo-randomly reads the memory contents and generates a checksum on these contents using the attester's key. However, this naïve approach is susceptible to a simple attack [32]. The attacker can modify the attestation routine such that instead of reading the sensor's memory contents, the routine reads the unmodified contents stored somewhere else by the attacker and computes a checksum on these contents. Since the routine is forced to read the unmodified memory contents, the checksum will be valid, and the attacker will be able to conceal his changes.

One important observation is that in this case, in order to generate a valid checksum, the attacker's modified attestation routine has to check before every memory read whether the current memory address belongs to the the modified portion of the memory by inserting `if` (or similar) statements. The attacker has to use static analysis techniques (that analyze program binary without executing it) to understand the routine and insert `if` statements within the routine, which also increases the execution time of the routine. The attester can use this fact to detect the attacker's modifications by measuring the actual time taken by the routine (running on the sensor) to generate the checksum and comparing it with the expected execution time. If the time taken to generate the checksum is greater than the expected time, the attester proclaims that the sensor is compromised. This approach was introduced by SWATT [32].

However, while performing software-based attestation over the network, the detection mechanism cannot be completely dependent on such minute execution delays, as the network and the current execution state of the sensor can introduce some unforeseen delays resulting into inaccurate measurement of the execution time of the attestation routine, and, thus, resulting in false positives or false negatives. Therefore, in order to accurately measure the execution delay, the attester should be in physical contact with the sensor, which cannot be always possible in practice.

Contributions. The main contributions of the paper are summarized as follows.

- We present an approach for detecting malicious changes to the static memory contents of wireless sensors. The approach allows the attester to attest the memory contents of the sensors over the network without requiring physical contact with the sensors.
- Our approach is not dependent on precise measurements of execution timing delays to detect malicious changes to the memory of the sensors. Therefore, our approach is more practical and can be used in real-world scenarios.
- Finally, the approach presented in this paper does not require any hardware support. Thus, we do not add any additional cost or increase the size of the sensor. Further, our approach can be easily applied on legacy systems.

Scope of this paper. This paper is focused on designing software-based attestation techniques that are secure against the static analysis attacks described above. We do not require any tamper-proof hardware on the sensors.

This paper is not focused on addressing the following impersonation attack, as detecting this attack requires additional tamper-proof hardware on the sensor. Consider an adversary that controls two identical sensors, or one sensor and one powerful machine that emulates the sensor. The attacker then modifies the memory contents of one sensor and keeps the other sensor or the emulated sensor unmodified. When the modified sensor receives an attestation routine, it forwards the routine to the other unmodified sensor or the emulated sensor on which it gets executed. Since the routine is executed as if on the original unmodified sensor, it will return a valid checksum and the modifications will go undetected. This attack can be detected by authenticating the actual processor that executed the authentication routine. However, this requires additional tamper-proof hardware on the sensor, e.g., controlled physical random functions [12, 13].

Organization. The remainder of this paper is organized as follows. Section 2 describes our system assumptions, requirements, and the attacker model. In section 3, we present the basic building blocks that are used to construct the attestation routine. Section 4 explains our attestation mechanism in detail. Section 5 presents security analysis of our system and an extension to our basic mechanism. Related work is presented in section 6 and section 7 draws conclusions and outlines future work.

2 Assumptions, Threat Model, and Requirements

2.1 Assumptions

The base station is assumed to be secure and it will play the role of an *attester* in our discussion. In reality, any legitimate entity that shares a pairwise key with the sensor can be an attester. The communications between the base station and the sensors is secure using a pairwise key shared between them. We do not address denial of service attacks (DoS) in this paper. The attester knows the hardware architecture and the original memory contents of the sensors. We assume that the sensors do not have virtual memory, as an attacker can modify the memory map, distinguish between data loads and instruction loads as pointed out in [11], and evade our attestation. We argue that this assumption is reasonable, since state of the art micro-controllers do not have virtual memory support [2, 3]. The attester can communicate with all the sensors directly. We also assume that the attester can send a binary executable to the sensor and cause it to be executed (e.g. [18]).

2.2 Threat Model

We assume that if the sensor is compromised, then the attacker has complete read-write access to the sensor's memory contents, including cryptographic keys, and is able to modify the memory contents at will. Thus, he can perform any type of software based attack on the attestation routine including static analysis (resulting in modification) of the routine, or software emulation of a sensor on a sensor. However, we assume that the attacker cannot tamper with the hardware of the sensor. Detection of attacks that involve external resources (such as the impersonation attack described in section 1) requires hardware support and is considered to be out of scope. We assume that the attacker can perform a restricted form of collusion attack, which we call as the staging attack. We assume that the attacker can execute the attestation routine in stages. For example, a sensor with some modified portion of the memory can collude with the second sensor with a different modified portion of the memory. Each sensor runs the routine in such a way that it generates checksum on their respective un-modified memory and then combine their checksums in the end to generate a valid checksum. Finally, the attacker can perform passive attacks such as eavesdropping, and active attacks such as replaying packets.

2.3 Requirements

The attestation procedure should satisfy the following requirements.

- **Resistance to Replay**: The attacker should not able to send a valid checksum to the verifier by simply replying previous valid results.
- **Resistance to Prediction**: The attacker should not be able to predict the next attestation routine. If the attacker can successfully predict the next attestation routine, then he can pre-compute the checksum.

- **Resistance to static analysis**: The attacker should not be able to successfully analyze the code by using static analysis techniques within the time period the attester waits for a response from the sensor. This requirement will prevent the attacker from predicting the sequence of memory reads as well as predicting the location of read instructions in the attestation routine.
- **Very loose dependence on execution time**: Since the attestation routine is sent over the network, it will be impossible for the attester to measure the actual execution time of the attestation routine. Therefore, the detection mechanism should not be dependent on the precise measurement running time of the attestation routine.
- **Complete memory coverage**: To detect even small memory changes, the attestation routine should read every memory location.
- **Efficient construction**: The attestation routine should be as small as possible to reduce bandwidth consumption and should be as efficient as possible to consume less battery power. Further, the attestation routine should not introduce any new vulnerability in the system.

3 Building Blocks

In order to prevent the attacks mentioned in Section 2.2, the attestation code will make use of the following building blocks. These constructs, which are described below, include randomization, encryption, obfuscation, and self-modifying code. These are not employed to provide unbreakable security, but rather they are used to make the aforementioned attacks infeasible to be carried out using a sensor's limited resources.

Randomization. The routine that is sent to the sensor to perform the attestation should be different each time. If the routine is different, in some random fashion, and the results of the attestation calculation are dependent on the specific version that is being run, then a previous version of the code could not be analyzed offline and reused later. Thus, the attacker is forced to perform the static analysis of the binary in an online fashion: the attacker needs to analyze and modify the routine and then execute it to return the result.

Encryption. The next construct that we use in the construction of the attestation code is encryption. We will make use of a simple encryption scheme (XOR each word with a random value) to prevent static analysis of the code directly. The attacker will thus need to first attack the decryption code in order to break the encryption of the remaining code. The encryption schemes are not meant to be secure in the traditional sense, but rather are aimed at adding complexity to the disassembly of the code. This technique has been explored previously in the field of software tamper resistance [4].

Self-Modifying Code. In addition, we use self-modifying code in the attestation program. Without this construct, an attacker could avoid doing the full static analysis of the code and just search for all memory read statements in the program. By doing this, the attacker can simply place conditional offsets before each read statement. However, if the reads are regenerated and rewritten in a different memory locations, then the attacker must first analyze the code that performs these writes. Without doing so the attacker could not reliably redirect the targets of these memory reads. The usage of this construct is explained further in Section 4.2 and has been proposed to strengthen operating system security [7].

Opaque Predicates and Pointer Aliasing. With the previous construct in place, the attacker is forced to analyze the entire program that it is sent. Thus we also add constructs to further complicate the task of static analysis as much as possible. For this purpose we use traditional obfuscation constructs, namely opaque predicates and pointer aliasing. Opaque predicates are predicates that always evaluate to either true or false, regardless of the input to the condition, yet it is very difficult to determine which branch will be taken each time, or even to determine whether this conditional is actually unconditional. Constructions of this type have been previously discussed in obfuscation literature [10, 8, 9]. One of the most promising constructions of opaque predicates is the use of pointer aliasing [28, 37] and performing data flow analysis of aliased pointers is known to be an NP-hard problem [17, 24, 30].

Junk Instructions. The use of junk or fake instructions can be combined with the opaque predicates described above to further confuse static analysis and disassembly [26]. Some of these junk instructions can be partial instructions, which will confuse the disassembly and thus hinder static analysis. Other instructions will be full instructions, which will be used to misdirect the static analysis and waste its time and efforts.

4 Design of Attestation Procedure

We now bring together all the building blocks described previously in Section 3 and describe our scheme to perform the attestation. Throughout the description of our scheme, we use the word code and routine interchangeably to refer to the attestation routine sent to the sensor.

4.1 Overview

The base station generates the attestation code, which will be sent to the sensor. The code construction is described in Section 4.2. When sending the code to the sensor, the base station encrypts the code and appends a MAC of the encrypted code, and sends this to the sensor. Upon receiving this message, the sensor first verifies the MAC and then decrypts the attestation code. The sensor then copies this into its program memory and transfers execution control to it. The attestation code will run and calculate the results. Once the result is calculated, it is sent back to the base station (again, this message is encrypted and authenticated by the sensor with the key it shares with the base station). Since the base station knows the image of the sensor's program memory, it can also run the code to compute the expected result. If the returned value matches the base station's expected result, then the sensor is declared to be ok. If the result is incorrect or if the sensor does not respond with the timeout period δ, then the sensor is declared to be corrupted. The base station should wait for a timeout period T_{wait} equal to $(2 * r) + e + \Delta$, where r is the time required to send a message from base station to the sensor (one way), e is the expected execution time of the attestation code, and Δ is a system parameter that indicates expected delay in the response due to network jitters, etc.

4.2 Attestation Code Construction

The high level construction of the code is as follows: the attestation code will generate random numbers (within the range of the sensor's memory that is to be attested) and reads the data at those memory locations. Those values will be hashed together incrementally (thus the order in which the data is read influences the final outcome of the attestation code). The code will also include each of the constructs mentioned in section 3, in order to prevent an attacker from modifying the code to avoid detection. This section will describe in detail how the code will be constructed to use those constructs.

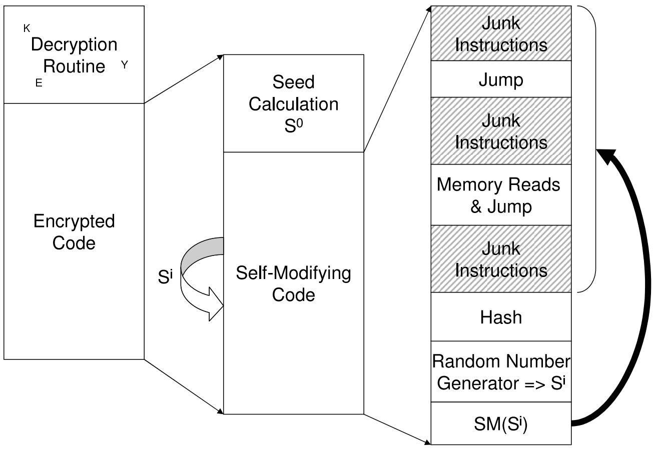

Fig. 1. Overall Structure

There are three main components in the code: the *seed calculation*, the *memory reads*, and the *hash computation*. For simplicity, we will describe the construction of the code with the assumption that each of these three parts are located in contiguous sections of the code and in the order described. However, the components can easily be interleaved with each other, with appropriate jumps between the different sections (obscured by opaque predicates). Figure 1 illustrates the construction of the attestation code.

Before discussing each component, we first describe the process of encryption. Not only is each component encrypted with a random key, but the entire attestation routine is also encrypted. Along with the encrypted code is the corresponding decryption routine. This will include the calculation of the key value, and the code to perform the decryption. The key will be located somewhere in the sensor's memory (either within the decryption routine itself, in some dead code space in the attestation routine, or using known portions of the sensors program memory), and so it will not be overly difficult for the attacker to discover the key. This discovery can be delayed, however, through the use of opaque predicates. Thus, we couple the key calculation with the opaque predicates, to obscure the location of the key (or components of the key). Then, the key will be calculated, which can be done in a number of ways, and the specific method used will vary randomly between each attestation routine. Some example mechanisms include taking the XOR of two (random) immediate values, adding together values from two "random" memory locations in the sensor's program memory, following multiple pointer indirections to the key value (where the pointers point to locations in the attestation routine). There are many such possibilities, and a few will be chosen at random (where one is the true method and its identity is hidden by the opaque predicates).

Seed Calculation. The first component of the code is the calculation of the seed, denoted as S^0 in Figure 1, which is used to initialize the pseudo-random number generator. As this value determines the order in which the memory contents are read, it will be in the attacker's best interest to leave this section of the code unmodified. Rather the attacker will need to determine a priori, through static analysis, the value of the seed. To prevent this, the seed calculation section is encrypted with a random encryption routine as described above. Also, the calculation of the seed will be done in the same manner as key calculation as described above.

The next two parts, memory reads and hashing, will be a part of a loop, where a location is read, and then the value is added to the hash computation. This loop will execute "enough" times to provide good coverage of the sensor's program memory (thus reducing the ability of the attacker to evade detection by hiding in a very small section of memory).

Memory Reads. The portion of the code that performs the memory reads is of particular importance, since that is where the attacker will attempt to inject the offsets in order to evade detection. This portion of the code has three main components: the *initial jump*, the *read* instruction, and the *self-modification.* The read instruction is initially located at some random address within this component, and so the jump instruction simply jumps control to this instruction. This jump, however, is obscured by the use of opaque predicates. In addition, as there is dead-code space, some of which will appear to the attacker to be reachable through the opaque predicate, junk instructions are inserted (randomly) in this space, both to thwart disassembly (with partial instructions) and to distract the static analysis (with normal instructions, such as memory reads). Following the read instruction, which places the contents of the particular memory address in question into a register, control is jumped to the self-modification section. This section is responsible for a number of tasks. First it generates three pseudo-random numbers. The first is used as the seed to the next iteration of the routine, denoted in Figure 1 as S^i. It uses the second as the next address to be read (and thus must be within the target range of addresses). The third is used to relocate the read instruction. It does this by overwriting the current read instruction with a junk instruction (or leaving it as is), and writing the new memory read instruction into another place in the code section. It also must update the initial jump so that control will be properly transferred to the new read instruction in the next iteration. This action is depicted in Figure 1 as $SM(S^i)$.

The random numbers will be generated using the RC4 pseudo-random number generator, as is used in SWATT [32]. In order to provide ample coverage of the memory space, it will iterate $O(n \log n)$ times though the memory read loop (this was shown in [32] to be a sufficient number of iterations to provide high coverage of the memory space), where n refers to the number of memory locations to be read.

Hash Computation. Next, the hashing component updates the current computation of the hash with the value that was read in the previous step. Once the computation of the hash is complete (all memory addresses are read), the final value is returned to the base station. We use the same hashing mechanism as in [32].

Construction by Base Station. The last item to be considered is the construction of each attestation routine by the base station. The base station must generate each attestation routine differently, such that the probability of two sensors receiving the same attestation code is very low (also the probability of a single sensor receiving the same code more than once should be very low). Thus each version of the attestation code must be generated randomly. This is achieved in several ways. First, the construction of the opaque predicates is based on the pointer-aliasing construction described in [9]. In our construction, these structures will be stored in random locations in the attestation code, and thus the opaque predicates will be different for each attestation routine. In addition, the seed will be chosen randomly, and the method used to compute this value will be chosen randomly from a set of possible methods.

5 Discussion

5.1 Security Analysis

In this paper we have presented a scheme for software based attestation that can be used in wireless sensor networks. In this section we will provide a discussion on the security properties of our scheme against the attacks described in Section 2.2.

First, an attacker can simply replay a previously computed response, or he could “sniff” a response from another sensor. However, in this case the attacker would only be successful if the seed used in the current attestation challenge is the same as the seed in the previous attestation challenge. Since the seed is chosen uniformly at random, this would only occur with negligible probability.

Thus the attacker must attempt to defeat the code contained in the current attestation challenge. Our goal is to force the attacker to perform some level of time intensive computation (which would delay the response to the attestation challenge past the timeout period). We argue that static analysis, while currently impossible to prevent, is computation intensive and will cause a significant delay in the response of the (resource-limited) sensor to the attestation challenge.

The attacker, then, has several options available in which to attack the code. First, the attacker can have an old version of the attestation code already analyzed (done offline at some previous point in time) and appropriately modified to avoid detection. In order for the attacker to be able to use this version of the attestation routine, he must first get the seed from the new attestation code. However, the attestation code, as well as the seed computation component, is encrypted. Thus the attacker must first break the two encryption schemes, which consists of determining the key that is used. This is protected by the opaque predicates. Also, once the attacker breaks the encryption schemes, he must determine the value of the seed, which is protected in the same way as the encryption keys. In order to accomplish these tasks, the attacker must perform static analysis on the code.

The attacker might also try to modify the read instructions, in order to insert the conditional offsets to redirect the read to the unmodified copy of the sensor's original code. The attacker could also use these modifications to redirect the reads to a collaborating sensor where that portion of the memory is unmodified (as per the staging attack described in Section 2.2). This also requires the attacker to determine the value of two encryption keys. In addition, due to the self-modifying code, the attacker cannot simply insert the conditional offsets into the code, but must first analyze the self-modifying portion of the code. By doing this, the attacker can cause the code to regenerate not only the memory reads, but the conditional offsets as well. Otherwise, if the attacker simply inserts the code before the initial read, the attestation code will overwrite this conditional offset and it will be lost. Thus, to do this, the attacker must again perform static analysis on the code.

Finally, the attacker can execute the attestation code within an emulator. In order for this attack to succeed, the attacker would pause execution of the code at each memory read, and offset the memory address to be read to point to an unmodified copy of the original sensor code. Emulation, however, imposes an inevitable slowdown in the execution of the program, and can be as much as an order of magnitude slower, as shown in [20]. Instructions are no longer decoded in hardware but in software. Also, code must be executed to process each emulated instruction. As this code is not bound by I/O, the slowdown will be significant, and with an appropriate choice of a timeout period, the base station can detect such an attack.

5.2 Extension

In addition, an optional extension to our scheme can be utilized to make emulation (and also static analysis) more difficult for the attacker to perform. During initial program code installation on the sensor, any free space in the sensor's program memory will be filled with random values. These random values will be known by the base station and thus can be included in the memory that is attested. Thus, the only free space available for the attacker to store an unmodified copy of the sensor's original code would be in the data memory. This is effective for two reasons. First, the data memory is typically used to store current execution information, such as the program stack, and thus certain portions of it cannot be overwritten by the attacker (without causing the sensor to crash). This not only reduces the amount of available space (thus requiring the malicious code to be very efficient), but also requires the attacker to be careful where the unmodified copy of the sensor's original code can be stored. Second, the data memory is typically much smaller than the program memory [1, 3], and thus the size of the malicious code (which performs static analysis or emulation) must be small enough to fit within the data memory.

6 Related Work

Software tamper-resistance is a technique to construct a program that either cannot be modified or an modification can be detected. There have been a variety of proposed approaches for achieving tamper-resistance. In general requiring additional hardware support has been one direction taken for solving this problem. On the other hand software based techniques such as obfuscation can be employed.

A trusted platform is one which adequately guarantees the users that the hardware and software modules are operating as specified. The Trusted Computing Group (TCG) has proposed an architecture called Trusted Platform Module (TPM). The TPM hardware, which is accompanied by supporting software, is used establish and provide a platform of trust. Load-time attestation using TPM is explored in [31]. BIND [34] employs TCG to perform fine-grained attestation; that is, it does not attest the entire memory but only a specific piece of code. Secure processors to prevent software tampering have been proposed in [25, 39, 35, 40]. Copilot [19] is a co-processor based runtime memory attestation mechanism. These hardware based approaches are not a suitable solution in our setting as sensors are expected to be inexpensive, and additional secure hardware would be prohibitively expensive.

SWATT [32], is a scheme that has been proposed to verify the static contents and configuration settings of an embedded device. However, as discussed previously in Section 1 this approach is not suitable for our setting. Genuinity [20], is a technique to ascertain whether a remote machine is running a real hardware running the expected software environment or not. Subsequently, attacks on Genuinity were described in [33, 32]. However in [21], the authors claim that the attacks on Genuinity are not sufficient to defeat the system. Our approach is similar in concept to Genuinity, in that we also send an attestation program to the sensor. As noted in [21], intricate details that could be exploited in embedded devices are rare, hence we have adopted sending a tamper-resistant attestation routine to achieve our goal.

The majority of the work on software based tamper-resistance relies on *obfuscation*. The goal of an obfuscating transformation is to make static analysis and disassembly of the executable, for the purpose of making useful modifications to a program, difficult [37, 10, 8, 9, 28, 38]. Theoretical work on obfuscation has yielded interesting results [14, 5, 36, 29, 27], and has shown that in general, perfect obfuscation is impossible. Therefore, careful choice of obfuscation transformations is necessary. For example, in [26], the authors proposed using indirect jumps (via branch functions) for preventing disassembly. By analyzing the control flow graph of the program and exploiting statistical techniques, the authors of [22] were able to correctly identify a majority of the program instructions.

Program evolution [7] was proposed as a technique to defend against automated attacks on operating systems. Self-checksumming software tamper resistance has been proposed in [6, 16]. Recently [11], the authors have shown the inadequacies of [6, 16] and proposed a generic attack on checksumming based software tamper resistance. The attack presented in [11] relies on advanced processor nuances like memory hierarchy, virtual memory and TLB, which is currently not available in sensors [1, 3], and hence is not applicable to our approach.

Integrity Verification Kernel (IVK) [4] is a technique for constructing tamper resistant *software*, where the software (that needs to be attested) is "armored" by means of self-encryption and self-decryption at run-time, coupled with self-checking of its integrity. This, however, is inherently different from our goals (attesting memory). If IVK is included in the sensor's programs, the attacker can simply reprogram the sensor. Further, the attacker can run the IVK in an emulator and get the actual (unencrypted) binary. If the attacker succeeds in getting the binary in clear, the attacker can generate a valid checksum on modified code. In our scheme, the attestation routine has to send a valid checksum on all of the static memory contents within the timeout period T_{wait}. Further, since the routine is new for each attestation, even if the attacker breaks one attestation routine, he cannot generate the checksum for the next attestation routine.

7 Conclusion and Future Work

Software attestation in sensor networks is one of the most important security primitives. To the best of our knowledge this effort is the first to consider remote software based attestation in sensor networks. We have presented a scheme which achieves this goal by sending a checksumming routine to the sensor from the base station. This code is protected by the techniques of encryption, obfuscation and self-modifying code, so that an attacker is unable to return a valid response from a compromised sensor within the allowed time. In addition, our approach is software based, and does not require the addition of any extra hardware.

Future work includes implementing and evaluating the presented attestation procedure. We are currently exploring ways to efficiently send the attestation routine to execute it on the sensor. We plan to explore the Mica Mote platform [2] and TinyOS [15], as this platform is known to support in-network reprogramming [18]. Detailed experiments will be performed to measure the overhead imposed by the attestation routine on the sensor in terms of battery consumption, code size, and execution time. Tests will also be performed on simulated sensors that can be used to simulate a large sensor network. As part of the experiments, we plan to measure the expected runtime so that we can provide estimates on the amount of time that the base station will wait for a response. We will also study the effect of Δ on the security of the system. Finally, a detailed security analysis of the implemented program will be provided.

References

1. Atmel AVR 8-bit RISC processor. `http://www.atmel.com/atmel/products/prod23.htm`.
2. Mica2 series.`http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf`.
3. TI MSP-430 processor. `http://focus.ti.com/mcu/docs/techdocs.tsp?navSection=user_guides&templateId=5246&familyId=342`.
4. D. Aucsmith. Tamper resistant software. In *Proceedings of the First Information Hiding Workshop*, 1996.
5. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 1–18, London, UK, 2001. Springer-Verlag.
6. H. Chang and M. J. Atallah. Protecting software code by guards. In *DRM '01: Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*, pages 160–175, London, UK, 2002. Springer-Verlag.
7. F. Cohen. Operating system protection through program evolution. Computers and Security, 1993.
8. C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Technical Report 148, Department of Computer Science, University of Auckland, July 1997.
9. C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, San Diego, CA, Jan. 1998.
10. C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. In *IEEE Transactions on Software Engineering*, volume 28, pages 735–746, August 2002.
11. A. S. G. Wurster, P.C. van Oorschot. A generic attack on checksumming-based software tamper resistance. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.
12. B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Controlled Physical Random Functions. In *Proceedings of the 18th Annual Computer Security Conference*, December 2002.
13. B. L. P. Gassend. Physical random functions. Master's thesis, Massachusetts Institute of Technology, February 2003.
14. S. Hada. Zero-knowledge and code obfuscation. In *ASIACRYPT '00: Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security*, pages 443–457, London, UK, 2000. Springer-Verlag.

15. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, Cambridge, November 2000.
16. B. Horne, L. R. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *DRM '01: Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*, pages 141–159, London, UK, 2002. Springer-Verlag.
17. S. Horwitz. Precise flow-insensitive may-alias analysis is np-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, 1997.
18. J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *The First IEEE International Conference on Sensor and Ad hoc Communications and Networks*, October 2004.
19. N. L. P. Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 179–194, 2004.
20. R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *12th USENIX Security Symposium*, pages 295–310. USENIX Association, August 2003.
21. R. Kennell and L. H. Jamieson. An analysis of proposed attacks against genuinity tests. Technical report, Purdue University, 09 2004. CERIAS TR 2004-27.
22. C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of USENIX Security 2004*, pages 255–270, San Diego, CA, August 2004.
23. T. Kuennen. Small science will bring big changes to roads. `http://www.betterroads.com/articles/jul04a.htm`.
24. W. Landi and B. G. Ryder. Pointer-induced aliasing: a problem taxonomy. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 93–103. ACM Press, 1991.
25. D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 168–177, New York, NY, USA, 2000. ACM Press.
26. C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, New York, NY, USA, 2003. ACM Press.
27. B. Lynn, M. Prabhakaran, and A. Sahai. Positive results and techniques for obfuscation. In *EUROCRYPT '04*, 2004.
28. T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software tamper resistance based on the difficulty of interprocedural analysis, August 2002.
29. T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software obfuscation on a theoretical basis and its implementation. In *IEICE Transactions on Fundamentals*, volume E86-A, pages 176–186, January 2003.
30. G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.
31. R. Sailer, T. Jaeger, X. Zhang, and L. van Doorn. Attestation-based policy enforcement for remote access. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 308–317, New York, NY, USA, 2004. ACM Press.

32. A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Software-based Attestation for Embedded Devicesi. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.
33. U. Shankar, M. Chew, and J. Tygar. Side effects are not sufficient to authenticate software. In *13th USENIX Security Symposium*. USENIX Association, August 2004.
34. E. Shi, A. Perrig, and L. V. Doorn. Bind: A time-of-use attestation service for secure distributed systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.
35. G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171, New York, NY, USA, 2003. ACM Press.
36. N. P. Varnovsky and V. A. Zakharov. On the possibility of provably secure obfuscating programs. In *Ershov Memorial Conference*, pages 91–102, 2003.
37. C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, University of Virginia, Charlottesville, VA, USA, 2000.
38. G. Wroblewski. *General Method of Program Code Obfuscation*. PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.
39. J. Yang, Y. Zhang, and L. Gao. Fast secure processor for inhibiting software piracy and tampering. In *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 351, Washington, DC, USA, 2003. IEEE Computer Society.
40. X. Zhuang, T. Zhang, and S. Pande. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 72–84, New York, NY, USA, 2004. ACM Press.