Mechanisms to Provide Integrity in SCADA and PCS devices [*]

Aakash Shah
Carnegie Mellon University
aakashs@andrew.cmu.edu

Adrian Perrig
Carnegie Mellon University
adrian@ece.cmu.edu

Bruno Sinopoli
Carnegie Mellon University
brunos@ece.cmu.edu

Abstract

Supervisory Control and Data Acquisition (SCADA) systems control and monitor critical infrastructure such as natural gas, oil, water, waste-water, and electric power distribution and transmission systems. SCADA systems consist of a central control center connected to Remote Terminal Units (RTUs) which directly interface with sensors and actuators connected to the physical infrastructure. Most RTUs are not designed with security in mind and consequently are vulnerable to various attacks compromising their code integrity. In this paper, we propose the use of software-only schemes that can be implemented on RTUs to provide verification of code integrity, untampered code execution and secure code updates.

1 Introduction

Supervisory Control And Data Acquisition (SCADA) systems are Process Control Systems (PCS) that monitor and control critical infrastructure such as the electric power, natural gas, oil, water and waste-water distribution and transmission systems. They are distributed systems consisting of a central master station and human machine interface (HMI), Remote Terminal Units (RTUs) connected to sensors and actuators, and a communications infrastructure. SCADA systems have historically been designed without any information security considerations. The use of private networks and proprietary protocols has provided some level of "security by obscurity" in the past. Clearly, this is not sufficient to secure systems that control critical infrastructure. Nowadays, SCADA systems are increasingly being connected to the corporate IT infrastructure and Internet, making them vulnerable to a remote attacker. It is imperative that these systems be secured as their compromise could have severe consequences.

Many steps need to be taken to properly secure SCADA systems. We discuss a few of these steps. Appropriate access control mechanisms need to be implemented on the SCADA master and RTUs. Technologies such as firewalls and intrusion detection/prevention systems need to be deployed to prevent unauthorized access to the SCADA system. SCADA systems have non-existent communications security. Strong message authentication is required on the communications channels between the SCADA master and RTUs. Encryption should be used to provide secrecy.

Prior work attempts to address these issues. The American Gas Association [1] describe a protocol for serial SCADA communications designed to defend against a Dolev-Yao adversary. The IEC 62351 standard [3] and the National SCADA Testbed group [8] propose authentication protocols for SCADA. Some commercial SCADA devices now allow for SSL connections [9]. Cunningham et al. [5] present tools to verify access policy implementations and a model-based intrusion detection system for SCADA systems.

However, despite many of these security mechanisms, it may still be possible for an attacker to compromise the RTUs. A concern in SCADA security is that an attacker would gain remote access to a large set of RTUs in the SCADA network and modify their software to launch a coordinated attack against the critical infrastructure. Such an attack can be effective in disabling the critical infrastructure in a region despite any inherent redundancies in the physical infrastructure. If the attack spans multiple utilities then the inter-dependency of the critical infrastructures could make the consequences of the attack worse. It is even possible for the attacker to program the RTUs to report correct data to the central control center, thus effectively hiding the attack.

In order to prevent such an attack, a SCADA operator should be able to detect if malicious software has been installed on the RTUs. In fact, the operator should be able to verify the integrity of the code on the RTU, perform secure code updates and ensure untampered execution of code. In this paper, we use tamper evident software primitives to present such a solution and contribute an implementation on a commercial SCADA RTU. Such tamper evident software primitives are discussed in prior work [6, 10–12].

At the core of these primitives lies a self-checking verification function that computes a checksum over its own instructions. A challenge-response protocol is employed between a trusted external verifier and the RTU. The external verifier sends a random challenge to the RTU. The verification function running on the RTU computes a checksum over its own instructions and returns the result to the external verifier. The

[*]This research was supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, TRUST (Team for Research in Ubiquitous Secure Technology), and the Gas Technology Institute. 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, CMU, GTI or the U.S. Government or any of its agencies.

checksum computation is designed in way such that if an adversary tampers with this function either the checksum will be incorrect or there will be a noticeable increase in the computation time. Thus, if the external verifier receives the correct checksum within the expected time, it can be sure that the verification function code on the device is unaltered.

The verification function also includes a cryptographic hashing function. Once the integrity of the verification function has been verified, a hash of the RTU's memory can be computed. The external verifier can compare this hash to the expected hash to ensure that the device has not been modified. Alternatively, a hash may be computed over a known executable to ensure that it has not been modified and then the hash function may invoke this executable in a way that ensures untampered execution.

Our proposed solution has several advantages. It provides strong guarantees regarding the integrity of the RTU's memory and the code running on the device. The RTUs represent a significant investment on part of the utilities and consequently any solution needs to be low-cost. Our software-only solution does not require any additional hardware and thus allows for a low-cost solution. The RTUs are hardened devices with lifetimes up to 30 years and hence there are many legacy RTUs in production systems currently. Our solution can be applied to present and legacy RTUs, thus allowing for a more secure SCADA system.

This paper is structured as follows. We discuss related work in Section 2. We present our assumptions and threat model in Sections 3 and 4 respectively. We provide an overview of an architecture independent implementation in Section 5 and discuss our implementation in Section 6. We discuss some practical considerations in Section 7. We discuss future work and conclusions in Section 8.

2 Related Work

The Trusted Computing Group (TCG) develops specifications for trusted computing and has designed the Trusted Platform Module (TPM), a tamper-evident chip that allows for a way to verify platform information by a series of attestations [13]. The TPM allows a verifier to obtain a guarantee of what code was loaded into system memory initially. However, a hardware based solution cannot be applied to legacy RTUs. Also, the TPM cannot be updated in software and consequently, the only way to modify them is to replace the hardware. The software-only solution described in this paper does not require any hardware extensions.

3 Assumptions

We assume that the external verifier (e.g., the SCADA control center) knows the exact hardware configuration of the RTU including the CPU model, CPU clock speed and the memory latency. We assume that the hardware of the SCADA RTU is not malicious and that it matches manufacturer specification. We assume that the CPU is not overclocked. We assume that the RTU has a single CPU without virtualization support. We assume that the RTU cannot access a faster computing platform (proxy) to perform computation on its behalf. We assume that the communications channel between the external verifier and RTU provides message-origin authentication i.e. the external verifier is guaranteed that packets are coming from the RTU.

4 Threat Model

We assume that the adversary has complete control over the software on the RTU including the OS. However, the adversary cannot modify the firmware of any peripherals to perform malicious DMA writes to the memory region containing the executable. We assume that the adversary does not physically modify the hardware of the device. This assumption may seem naive as most RTUs do not have strong physical security and are generally secured by a lock. Therefore, it is possible for an attacker to gain access to a particular substation and, replace or modify a RTU to allow for faster computation time. However, we assume that doing so at multiple substations is very hard and also dramatically increases the chances of detection. We are most concerned with remote attacks. Depending on the SCADA architecture, the adversary may access the RTUs directly over the Internet or by compromising devices in the SCADA control center. However, we assume that there is at least one trusted device at the SCADA control center that acts as the external verifier and cannot be compromised by the adversary. We assume that the adversary has control over the communications media and can add delays to the response of the checksum function to generate false positives, effectively creating a denial of service. However, this is no different than the adversary "cutting the wire" and therefore we do not address this issue.

5 Overview

In this section we discuss mechanisms to provide verification of code integrity, untampered execution and secure code updates on a SCADA RTU. Such mechanisms are discussed in detail in prior work [6, 10–12]. At the core of these schemes lies a self-checking verification function that can guarantee its own integrity by computing a checksum over its own instructions. The verification function thus dynamically establishes a trusted base or dynamic root of trust on the RTU. This dynamic root of trust can then be used to verify other aspects of the system such as code integrity. We present an architecture independent design of the checksum code in this section.

5.1 Verification Function Design

The design of our verification function is based on the Pioneer primitive [12]. Figure 1 provides an overview of the verification function. The verification function consists of three main components: the checksum function, send function and the hash function.

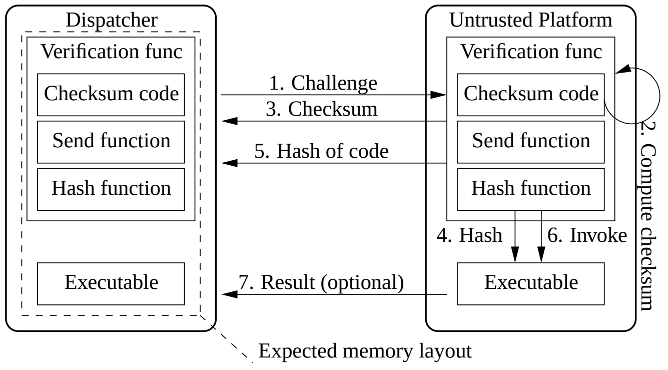

Figure 1: Verification function overview. Numbers represent temporal order of events.

Checksum Function. The checksum function computes a checksum over the entire verification function and sets up an environment in which the send function, the hash function, and the executable are guaranteed to run untampered by any malicious software on the RTU [12]. The checksum function needs to be designed such that even if a single byte of the verification function is modified, the checksum will be different. A correct checksum assures the external verifier that the code has not been modified. However, an adversary could presumably modify the verification function, and calculate the checksum over a valid copy of the verification function code, thus generating the correct checksum. In order to prevent this, the checksum is designed in way that any such modification would add considerably to the running time of the checksum function. Then, a correct checksum obtained within the expected amount of time guarantees that the verification function has not been modified and that there is an environment for untampered execution on the RTU.

Hash Function. A cryptographic hash function that is second preimage resistant (e.g., SHA-1) is used to perform the integrity measurement of the executable. A random nonce received from the external verifier and code for the executable are hashed and the resulting digest is returned to the external verifier. The external verifier can compare this digest to the expected one to ensure that the executable has not been modified. The hash function proceeds to invoke the executable when it is done.

Send Function. The send function sends the checksum and hash digest to the external verifier.

5.2 Required properties of checksum code

The checksum code has to be constructed such that any tampering of the verification function either results in a incorrect checksum or causes a noticeable delay in the checksum computation. We briefly discuss the required properties of the checksum code below. Seshadri et al. [12] provide a detailed description of these properties.

Time-optimal implementation. The checksum code needs to be the checksum code sequence with the fastest running time, otherwise the adversary could use a faster implementation of the checksum function and use the saved time to forge the checksum. To achieve a time-optimal implementation we use simple instructions such as “add” and “xor” that cannot be implemented faster or with fewer operations. Also, the checksum code is structured as code blocks such that the operations in one code block depend on previous blocks.

Instruction sequencing to eliminate empty issue slots. We arrange the instruction sequence of the checksum code so that the processor issue logic always has a sufficient number of issuable instructions for every clock cycle.

CPU state inputs. We incorporate CPU state inputs such as the Program Counter (PC) and the data pointer in the checksum function to defend against memory copy attacks discussed in prior work [12]. In a memory copy attack the adversary modifies the checksum function such that the checksum is generated over a good copy of the verification function code stored elsewhere in memory.

Iterative Checksum code. Iterative checksum code allows for a constant-time overhead per iteration, thus allowing for a noticeable delay in the checksum computation time by running the checksum for a large number of iterations.

Strongly-ordered checksum function. We use a strongly-ordered checksum function to prevent parallelization. A strongly-ordered checksum function is a function whose output differs with a high probability if the operations are evaluated out of order. A strongly ordered function requires the adversary to perform the same operations on the same data in the same sequence as the original function to obtain the correct result. We use a strongly ordered checksum function consisting of alternating “add” and “xor” instructions. This prevents parallelization, as at any step of the computation the current value is needed to compute the succeeding values.

Small code size. The checksum code must fit in the processor cache to achieve a deterministic execution time. Also, the relative per iteration overhead increases with a smaller checksum loop.

Low variance of execution time. We achieve this property by ensuring that the checksum code is run uninterrupted, both the verification code and checksum code are small enough to fit in the L1 data and instruction caches, respectively, and having sufficient issuable instructions for any CPU cycle.

Keyed-checksum. To prevent the adversary from precomputing the checksum, and to prevent replaying of old checksum values, the checksum depends on a random challenge sent by the external verifier.

Pseudo-random memory traversal. We use pseudo-random memory traversal to protect against the data substitution attack discussed in prior work [10, 12]. In the data substitution attack, the adversary modifies a certain memory region containing the verification function code and redirects memory reads for this region to another location containing a correct copy of the data.

5.3 Execution Environment for Untampered Code Execution

We discuss how the checksum function can be used to setup an untampered execution environment for the hash function, the send function and an executable.

We ensure that the checksum function runs at the highest privilege level and that all maskable interrupts are turned off. We include the flags register that contains interrupt enable/disable flags in the checksum computation to ensure that an adversary running a modified checksum function at lower privilege levels will generate the incorrect checksum.

To ensure that the hash function and the executable will run untampered, we have to guarantee that the the exception handlers and the handlers for the non-maskable interrupts are not malicious. To achieve this, we replace the existing exception handlers and handlers for non-maskable interrupts with our own handlers that return immediately. This ensures that the send function, hash function and executable run uninterrupted in an untampered environment.

5.4 Performing Secure Code Updates

The untampered code execution environment can be used to ensure secure code updates. Seshadri et al. discuss such a scheme for sensor networks [11]. The untampered code execution can be used to execute the update executable to ensure that no malicious code can interfere with its execution.

6 Implementation

In the following section we discuss our implementation on a commercial SCADA RTU. We implemented a verification function that provides a mechanism for software based code attestation and verifiable code execution based on the Pioneer primitive discussed by Seshadri et al. [12]. We used the SCADAPack 350 RTU made by Control Microsystems for our implementation. The SCADAPack 350 is a programmable logic controller geared towards the oil and gas, water, wastewater and electric utilities. It has a simple architecture and can be programmed using C/C++ or ladder logic.

6.1 Architecture Overview

The SCADAPack 350 uses a 32-bit ARM7TMDI processor with a 32 MHz clock [4]. The processor also has two microcontroller coprocessors with 20 MHz clocks. ARM CPUs use a Von Neumann architecture where data and instructions reside in the same memory. The SCADAPack 350 has 16 MBs of Flash ROM, 4 MBs of CMOS RAM and 4 KBs of EEPROM. It has 3 serial ports, 2 USB ports and 1 Ethernet port. It optionally comes with a wireless spread spectrum radio. The device runs VxWorks 5.5 as its operating system.

The ARM7TMDI core is a member of the ARM family of general purpose 32-bit RISC microcontrollers [2]. It implements two instruction sets: the 32 bit ARM and the 16 bit Thumb instruction set. It has seven operating modes: User, FIQ, IRQ, Supervisor, Abort, Undefined and System. While the processor has 37 total registers, a subset of the registers are “banked” across the operating modes. Each mode can access 16 general-purpose registers (including the program counter, link register and stack pointer) and the Current Process Status Register (CPSR). All operating modes except the user and system modes also have access to the Saved Process Status Register (SPSR).

6.2 Design of Checksum Code

We base the design of our checksum code on the PRISM implementation [6] as it is also geared towards an ARM based architecture.

Our checksum function must provide the properties discussed in Section 5.2. Any tampering should generate an incorrect checksum or lead to a noticeable computation delay. The checksum function takes a 68 byte input that is generated from the challenge provided by the external verifier. The checksum function computes the checksum over the memory region that the verification function resides in and returns a 68 byte output.

The checksum function is a time-optimal iterative function that uses all general-purpose registers and generates a 68 byte checksum which we represent as a vector of seventeen 32-bit checksum pieces. The use of all the general-purpose registers ensures that an adversary has no free registers at its disposal. Each checksum piece is stored in one register. Before the checksum loop begins, the checksum is initialized with the random challenge obtained from the external verifier, preventing the adversary from pre-computing the checksum. Each iteration of the checksum code performs the following steps:

Step 1. We use a 32-bit T-function [7] to generate a pseudo-random number. Adding the pseudo-random number to the address where the verification function is loaded ensures that the attacker cannot predict the memory locations being accessed. We use the following T-function, x = x + $(x^2 \vee 5) \bmod 2^n$. The initial x value is obtained from the random challenge issued by the external verifier.

Step 2. A 32-bit word is read from the computed memory address.

Step 3. One checksum piece is updated based on all other running checksum piece values, the pseudo-random memory address read, memory word read and other state variables.

Step 4. The state variables are updated and steps 1-4 are repeated for a different checksum piece.

Figure 2(a) shows the checksum function pseudocode.

```
//Input: Number of iterations $y$ of the checksum function,
//       Random challenge to initialize checksum pieces
//Output: Checksum $C$
//             $daddr$ - address of current memory access
//             $x$ - value of T-function
//             $status$ - CPSR register
//             $saddr$ - Address where verification function is loaded
for $l = y$ to 0 do
    //T function updates $x$ where $0 \leq x \leq 2^n$
    $x \leftarrow x + (x^2 \vee 5) \bmod 2^n$
    //Modifies $daddr$ based on the $saddr$ and $x$ from T function
    $daddr \leftarrow saddr + x[31:22]$
    //Read from memory address $daddr$, modify checksum.
    //Let $C$ be the checksum vector and $j$ be the current index.
    $C_j \leftarrow C_j + C_{j-1} \oplus C_{j-2} + C_{j-3} \oplus C_{j-4} + C_{j-5} \oplus C_{j-6} + C_{j-7} \oplus C_{j-8} +$
    $PC \oplus daddr + mem[daddr] \oplus status + l$
    $C_j \leftarrow rotate\_right(C_j) \oplus x$
    //Update checksum index
    $j \leftarrow (j+1) mod 9$
end for
```

(a) Checksum Function Pseudocode

```
Assembly Instructions
mul r11, r10, r10
add r1, r1, r0
eor r1, r1, r8
orr r11, r10, #0x5
adds r10, r10, r11
mov r11, #0x12, 12
add r11, r11, r10, LSR #22
add r1, r1, r7
and r11, r11, #0xfffffffc
ldr r12, [r11]
eor r1, r1, r6
add r1, r1, r5
eor r1, r1, r4
add r1, r1, r3
eor r1, r1, r2
add r1, r1, r15
eor r1, r1, r11
add r1, r1, r12
eor r1, r1, r12
mrs r12, cpsr
add r1, r1, r9
eor r1, r10, r1, ROR #1
subs r9, r9, #1
```

(b) Checksum Assembly Code. r1 is the current checksum register.

Figure 2: Implementation of checksum function.

6.3 Checksum Implementation

We run the verification function as an application in Supervisor mode. The checksum function is written in inline assembly while the rest of the verification function is written in C.

Figures 2(a) and 2(b) show the checksum function pseudocode and assembly code, respectively. Our checksum computation includes the previous nine checksum pieces, stored in registers r0 through r8, the program counter (r15), CPSR, the pseudo-random memory address, memory word read and the current iteration number. The checksum piece computation is a series of alternating "add" and "xor" instructions to allow for a strongly ordered function. We also rotate the running checksum piece to prevent an attack discussed by Seshadri et al. [10].

6.4 Empirical Analysis of Attack Overhead

In this section we discuss how any attack against the verification function will lead to a noticeable overhead in the checksum computation.

Other than hardware attacks, the best known attacks against the verification function attempt to forge the checksum in software. Such attacks can be classified into memory copy attacks or data substitution attacks. We show that such attacks incur a significant overhead in the checksum computation.

Since we use all general-purpose registers, the attacker has no free registers to use in its attack. The only way the attacker can obtain free registers is by storing some of the checksum state in memory or use the banked registers. However, accessing memory is slow and we show that this leads to a noticeable overhead in the checksum computation. Using the banked registers requires that the adversary add at least 6 instructions in assembly: 3 to switch into a different mode and 3 to switch back. If the additional 6 instructions do not add enough delay in computation, the checksum can be designed to use all the banked registers, ensuring that the adversary cannot use them.

To implement any attack the attacker may need to load immediate values (e.g. a specific memory address). The attacker can load this value from memory but this is slow. The attacker could encode the immediate operand within the instruction. However, in the ARM architecture immediate operands are limited to 8 bits. The attacker can use the barrel shifter to shift the 8 bit value in the same instruction cycle, but this still limits the number of 32-bit immediate values achievable in a single instruction. The attacker would need to add an offset to get the 32-bit number it desires. This requires that the attacker have a free register which we show to add overhead to the computation.

The attacker may implement its modified checksum loop using the Thumb instruction set in order to get higher code density. However, this would come at the expense of performance and the computation time would increase considerably. Also, including the CPSR in the checksum prevents the attacker from switching to the Thumb instruction mode.

The attacker may use the two co-processors present in the ARM7TDMI chip to get a faster checksum computation. We ensure that the checksum code is non-parallelizable by using a strongly-ordered checksum calculation and having each checksum piece depend on previous checksum pieces. Thus, the attacker would only gain an advantage if the co-processors were faster than the core ARM processor, which is not the case.

We defend against the memory copy attack by including the PC and the pseudo-random memory address in the checksum piece calculation. In order to implement a memory copy attack, the attacker has to forge either the PC or the memory address. In order to forge the PC the attacker has to use an immediate or load the value from memory. To forge the random offset of the memory address being read, the attacker has to read from memory. Both cases lead to a slow down in the checksum computation. However, the ARM instruction set allows one to load data from an offset from a memory location. The attacker can place an unmodified copy of the verification function at a fixed offset from the modified version. Then instead of loading data from the modified memory locations, the attacker can always load from the unmodified version of the verification function. A self-modifying checksum function can be used to prevent this attack. We do not use a self-modifying checksum function in the current implementation.

We defend against the data substitution attack in several ways. We access memory locations in a pseudo-random manner. This ensures that the attacker must check to see if the memory location being checked has been modified and then read from an alternate memory location. This requires the attacker to add a compare instruction to the checksum loop. This has two problems. First, the attacker needs a register that contains the memory address of the malicious code to perform the compare or use an immediate value. Second, the compare instruction modifies the flags in CPSR which is included in the checksum computation, thus requiring the attacker to save the original CPSR.

We measured the computational overhead of three different attack scenarios - data substitution attack, using the ARM banked registers and the memory copy attack. In the data substitution attack we modified the contents of one memory location and added the extra "if" statement to ensure that if the modified memory location is about to be accessed then the memory contents are loaded from an alternate memory location. In the banked register scenario, we simulate the switch to and from a different processor mode by adding six No OPeration instructions (NOP) in the checksum loop. We simulate the memory copy attack by adding appropriate load and store instructions to the checksum loop. We run the checksum loop for 16.2 million iterations in each case and compare the results to that of the unmodified checksum function. A dedicated serial link connected the SCADA RTU and external verifier. Figure 3 shows the results of our experiments. Note that the variance in the computation time for each scenario is negligible. Also, note that the banked register attack, memory copy attack and data substitution attack led to a 25%, 36% and 51% increase in computation time, respectively.

6.5 Untampered Execution Environment

We ensure an untampered execution environment by masking all interrupts and running at the highest priority. We include the CPSR in our checksum to ensure that an attacker cannot

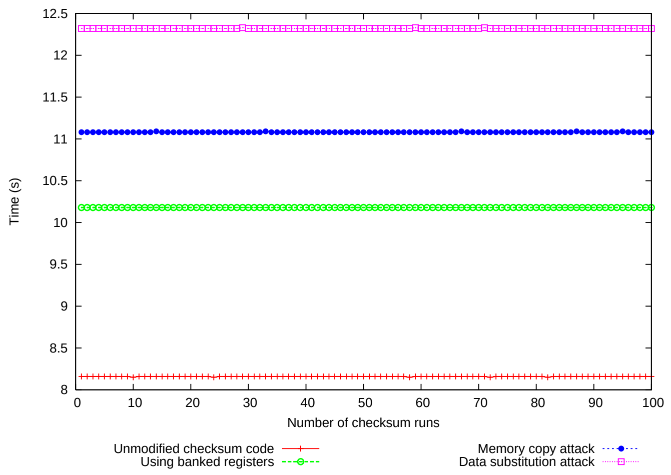

Figure 3: Checksum Computation Time for Various Tests.

switch to different processor mode. We do not replace interrupt handlers with trusted handlers in this implementation. Once the checksum function returns, a hash is computed of the executable and sent to the external verifier, interrupts relevant to the executable are enabled and the hash function invokes the executable. Similarly, the hash function is also used to compute a hash of memory to ensure the integrity of the RTU memory.

7 Practical Considerations

We discuss some of the practical considerations related to our work below.

Most SCADA RTUs have a simple architecture. This allows for a straightforward implementation of the verification function. The SCADA manufacturers are best suited to implement the verification function as they have an intimate knowledge of their architecture and the development tools necessary to modify the kernel. The external verifier can be a trusted device in the central control center. A SCADA operator must be responsible for keeping a updated copy of the RTU image on the external verifier.

There are hundreds of different SCADA manufacturers and this could require unique implementations for different SCADA manufacturer devices. However, different manufacturers may use common architectures (e.g., ARM) and existing designs and implementations for these architectures can significantly reduce the development time.

There is a concern that the communications infrastructure may add delays to the challenge-response protocol between the external verifier and the SCADA remote field device. In order to avoid false positives[1] in detecting malicious code, the threshold for detection must be increased to account for any

[1]Positive refers to the event that malicious code is detected.

such delays by running the checksum function for more iterations. This can be achieved by generating baseline figures for the delays on the communications channels. In cases where the variance in communication delays is high, the checksum function can be executed multiple times to ensure that there are no false positives. It is important to note that this primarily applies to PCS systems where the field devices reside in a remote location. For example, the field devices for a chemical plant PCS may be stored locally and will not have a large variance in communication delays.

Most SCADA systems are real-time distributed systems that are constantly running. The real-time application must be taken offline as running the checksum function in parallel could affect the process. If the real-time application allows it, it may be possible to stop the real-time application once a day for maintenance and run the verification function. However, in cases where the real-time application on the field device cannot be taken offline, redundancy must be used. A secondary field device must be brought online as the primary one is taken offline to run the verification function.

Another concern is the frequency of checks on the field device. It may be sufficient to run the verification function once a day to ensure that the field device software has not been modified. However, if the verification function is run on a fixed schedule then the attacker can take advantage of this and ensure that there is no malicious code on the device when the verification function is run and reload the malicious code right after the verification function finishes.

We assume message-origin authentication between the external verifier and the remote field device. However, most SCADA communications channels lack any form of authentication. If the attacker has control over the communications channel, it can launch a proxy attack where it computes the checksum on a second device loaded with good software and send the external verifier the correct checksum. Message authentication is necessary to prevent this attack.

Many SCADA systems use a cascaded architecture where RTUs are chained together. In such cases, the external verifier may be able to run the verification function on the RTU it is directly connected to, which in turn can run the verification function on the chained RTU. The checksum and hash values can be sent back to the original external verifier.

8 Conclusion and Future Work

We show how verification of code integrity, untampered code execution and secure updates can be achieved in SCADA and PCS field devices. We contribute an implementation on a commercial SCADA RTU, discuss the practical implications and demonstrate how such a scheme can be used to provide security in SCADA systems.

In future work, we intend to propose a scheme that uses the same tamper evident primitives to allow the SCADA master to obtain a guarantee that the analog and digital readings from the RTUs are authentic.

9 Acknowledgments

We would like to thank Arvind Seshadri and Tom Palarz for invaluable technical discussions and the Gas Technology Institute for loaning us equipment.

References

[1] AGA 12, Part 2 draft. *American Gas Association*, 2006.

[2] ARM. *ARM7TDMI Technical Reference Manual*.

[3] IEC 62351. *Power systems management and associated information exchange - Data and communications security*.

[4] Control Microsystems. *SCADAPack 350 Hardware Manual*.

[5] R. Cunningham, S. Cheung et al. Securing Process Control Systems of Today and Tomorrow. In *First Annual IFIP WG 11.10 International Conference on Critical Infrastructure Protection*, March 2007.

[6] J. Franklin, M. Luk, A. Seshadri and A. Perrig. PRISM: Enabling Personal Verification of Code Integrity, Untampered Execution, and Trusted I/O on Legacy Systems. *CyLab*, Feb 2007.

[7] A. Klimov and A. Shamir. A new class of invertible mappings. In *Proceedings of International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2003.

[8] Visualization and Controls Program Peer Review. *National SCADA Test Bed*, Oct 2006.

[9] OSI Remote Information System. `http://www.osii.com/solutions/products/remote-telemetry.asp`.

[10] 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*, Oakland, California, May 2004.

[11] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. SCUBA: Secure Code Update By Attestation in Sensor Networks. In *ACM Workshop on Wireless Security (WiSe 2006)*, Los Angeles, CA, September 29, 2006.

[12] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying Integrity and Guaranteeing Execution of Code on Legacy Platforms. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, United Kingdom, October 2005.

[13] Trusted Computing Group. Trusted platform module main specification, Part 1: Design principles, Part 2: TPM structures, Part 3: Commands.