SBAP: Software-Based Attestation for Peripherals *

Yanlin Li, Jonathan M. McCune, and Adrian Perrig

CyLab, Carnegie Mellon University
4720 Forbes Avenue, Pittsburgh, PA, United States
{yanlli,jonmccune,perrig}@cmu.edu

Abstract. Recent research demonstrates that adversaries can inject malicious code into a peripheral's firmware during a firmware update, which can result in password leakage or even compromise of the whole host operating system. Therefore, it is desirable for a host system to be able to verify the firmware integrity of attached peripherals. Several software-based attestation techniques on embedded devices have been proposed as potentially enabling firmware verification. In this work, we propose a Software-Based Attestation technique for Peripherals that verifies the firmware integrity of a peripheral and detects malicious changes with a high probability, even in the face of recently proposed attacks. We implement and evaluate SBAP in an Apple Aluminum Keyboard and study the extent to which our scheme enhances the security properties of peripherals.

1 Introduction

Recent research shows that adversaries can subvert keyboards by injecting malicious code into a keyboard's firmware during firmware update [1]. The injected code can compromise users' privacy and safety, such as eavesdropping a user's bank account password or credit card number, or embedding a kernel-level rootkit into a clean re-installed operating system through some software vulnerabilities in the host operating system. Similar attacks can happen on other peripherals, such as a mice or a game controller. Peripheral manufacturers enable updating of firmware to fix firmware bugs. However, due to constrained computation and memory resources, the low-speed embedded microcontroller on many peripherals cannot verify complex cryptographic signatures or message authentication codes. Consequently adversaries can inject malicious code into peripheral firmware during a firmware update. Therefore, a legacy computer is potentially

* This research was supported in part by CyLab at Carnegie Mellon under grants DAAD19-02-1-0389 from the Army Research Office, and by grant NGIT2009100109 from the Northrop Grumman Information Technology Inc Cybersecurity Consortium. 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, CyLab, NGC, or the U.S. Government or any of its agencies.

under serious attacks due to vulnerabilities on widely used peripherals. We take the position that it is desirable for a host machine to verify the firmware integrity on peripherals.

Software-based attestation schemes on embedded systems [2–4] have been proposed as potentially enabling firmware verification, which enables an external trusted verifier to verify the firmware integrity on peripherals. However, recent research [5] suggests that it may be feasible to hide the malicious code from an attestation through a return-oriented attack or a compression attack. In addition, constrained computation and memory resources in the low-speed peripherals limit the implementation of the software-based solutions. For example, some software-based attestation schemes [2] require hardware multiplication units or a large amount of data memory that is not available on all peripherals, especially on low speed peripherals such as mice or keyboards. Therefore, peripheral firmware integrity verification remains an important challenge.

In this paper, we propose Software-Based Attestation for Peripherals (SBAP) to verify peripherals' firmware integrity. Similar to previous proposals, SBAP is based on a challenge-response protocol between a trusted verifier and an untrusted peripheral, and a predicted computation time constraint. It verifies the contents of both program and data memory on the peripheral and can detect any malicious changes with arbitrarily high probability, even in the face of recently-proposed attacks. In this paper, we make the following contributions:

1. We propose a software-only solution to verify the firmware integrity of peripherals, that can be implemented via a software upgrade to the peripherals and avoid a costly hardware upgrade.
2. We propose an approach to verify the code or data integrity on both program memory and data memory in peripherals that can prevent all known attacks.
3. We design, implement, and evaluate a prototype of SBAP using an Apple Aluminum Keyboard.

We organize the remainder of this paper as follows. In Section 2, we provide the background on software-based attestation and related attacks. Section 3 presents the problem definition, assumptions, and the attacker model. In Section 4, we describe the system design including the system architecture, attestation protocol and verification function design. Section 5 details our SBAP implementation on an Apple Aluminum Keyboard and Section 6 gives our experimental results. We discuss our work in Section 7 and describe related work in Section 8. Finally, we offer our conclusions and identify future work in Section 9.

2 Background

2.1 Software-based Attestation for Embedded Devices

SWATT. SoftWare-based ATTestation for embedded devices (SWATT) is based on a challenge-response protocol between a trusted verifier and an untrusted embedded device, and a predicted computation time constraint. First, the verifier

sends a random nonce to the embedded device. Using this nonce as a seed, a verification function in the embedded device computes a checksum over the entire memory contents and returns the checksum result to the verifier. The verifier has a copy of the expected memory contents of the embedded device, so it can verify the checksum result. Also, the verifier knows the exact hardware configuration of this untrusted embedded device, enabling the verifier to exactly predict the checksum computation time. Because the checksum function is well optimized, the presence of any malicious code in memory will either invalidate the checksum result or introduce a detectable time delay. Therefore, only the checksum result received within the expected time range is valid. During checksum computation, the checksum function reads memory in a pseudo-random traversal, thus preventing an attacker from precomputing the checksum result. SWATT requires that the embedded device can only communicate with the verifier during attestation. This prevents a malicious device from communicating with a faster machine to compute the checksum.

ICE. Indisputable Code Execution (ICE) sets up a dynamic root of trust in the untrusted device through a challenge-response protocol between a trusted verifier and an untrusted embedded device, and a predicted computation time constraint. The dynamic root of trust also sets up an untampered execution environment, which in turn is used to demonstrate verifiable code execution to the verifier. As in SWATT, the verifier first sends a random nonce to the untrusted device. Upon receiving the random nonce, the verification function in the untrusted device sets up an untampered execution environment. The verification function includes code to set up an ICE environment by disabling interrupts, a checksum function that computes a checksum over the contents of the verification function, a communication function (send function) that returns computation results to the verifier, and a hash function that computes a hash of the executable that will be invoked in the untampered environment. After checksum computation, the send function sends the checksum result to the verifier. As in SWATT, the verifier can verify the checksum result and predict the checksum computation duration. If the verifier receives the correct checksum within the expected time, the verifier obtains assurance that the untampered execution environment (dynamic root of trust) has been set up in the untrusted device. The send function invokes the hash function to compute a hash of the executable in the embedded device and sends the hash result to the verifier. Then the verification function invokes the executable on the untrusted device. Simultaneously, the verifier obtains the guarantee of the integrity of the executable through verifying the hash of the executable.

Discussion. Both ICE and SWATT implement code integrity verification through a software-only approach. However, there are several challenges to implement ICE on embedded devices. In ICE, CPU registers describing code location (e.g., the program counter) and interrupt status information are included in the checksum to confirm the intended code location and interrupt status. However, not all microcontrollers provide instructions to directly access the values of the PC

or interrupt status registers. For instance, on the CY7C63923 microcontroller [6] in the Apple Aluminum Keyboard, there is no instruction to read the value of the PC, although the CY7C63923 does provide instructions to access the Flag register containing interrupt status information. Moreover, the code size of verification function in ICE is larger than the code size of the checksum routine in SWATT. On some embedded devices, there are very constrained memory resources for the implementation of the verification function. For example, on an Apple Aluminum Keyboard, the size of Flash memory is only 8 KB and there is only about 1 KB of free Flash memory for the implementation of our verification function.

2.2 Attacks on Existing Proposals

Memory Copy and Memory Substitution Attack. In a memory copy attack, the attacker modifies the checksum code in program memory while keeping a correct copy of the original code in unused memory. In a memory substitution attack, the attacker keeps the correct code in the original memory location, but deploys the malicious code in unused memory. In both attacks, the malicious code computes the checksum when expected. To obtain the correct checksum result, the malicious code redirects the location of the memory read operation to the correct code. On common embedded devices, the values in empty program memory are constant (i.e., 0xFF, which is the uninitialized value of Flash memory). Thus, the malicious code can predict such constant values and use them during checksum computation. SWATT and ICE prevent a memory copy or memory substitution attack by reading the program memory in a pseudo-random fashion so that the malicious code cannot predict the memory address to read, and has to add additional instructions to check and redirect the memory address. Such operations result in a detectable time overhead. On a Harvard architecture embedded device, the read latency of data memory is much smaller than the read latency of program memory. Thus, the malicious code can minimize the computation overhead of a memory copy attack by having a copy of the original code in data memory and redirecting the location of checksum memory read operations to data memory instead of program memory.

Compression Attack. One important enabler of a memory copy or memory substitution attack is that the malicious code can remember or predict the constant values of empty memory during attestation. Therefore, Seshadri et al. [2] propose to fill the empty space of program memory with pseudo-random values and leave no available free space for attackers to make a memory copy or memory substitution attack. However, an attacker can still create free space through compressing the existing code on program memory. Some compression algorithms, such as the Canonical Huffman Encoding [7], can decompress the compressed stream from an arbitrary position. Thus, the malicious code can decompress the compressed steam on-the-fly during attestation and obtain the correct checksum result though the checksum code reads memory in a pseudo random traversal.

The decompression procedure causes a detectable computation overhead because of the complexity of the decompression algorithm.

Return Oriented Programming Attack. Return oriented programming (ROP) [8–10] performs computation on a system by executing several pieces of code that are terminated by a return instruction. These pieces of code are executed through well-controlled stack content. If there is sufficient existing binary code in the system, an adversary can execute arbitrary computations through a ROP attack without injecting any code, except for overwriting the stack with well-designed content. Castelluccia et al. [5] present that an adversary can use a ROP attack to protect malicious code from being detected by software-based attestation schemes. Briefly, the adversary code first saves a copy of the adversary code on data memory before attestation. Then the adversary code modifies the contents of data memory by embedding ROPs on the stack. Through these ROPs, the attacker erases all the malicious code in program memory and restores the original code before checksum computation. Then, during checksum computation, the contents of program memory are exactly as expected. After attestation, the attacker restores malicious code in program memory through an additional ROP. The ROP attack generates little computation overhead. For example, in the attack described by Castelluccia et al. [5], the computation overhead caused by a ROP attack is undetectable, only 0.3% of the expected checksum computation time.

Attack Analysis. As described above, an attacker can hide malicious code from an attestation through a compression attack or a ROP attack. However, both attacks modify the contents of data memory, by storing either malicious code or ROP data. Thus, a checksum function can detect such malicious changes by verifying the contents of both program memory and data memory. However, it is challenging to verify the contents of data memory, since the content is unpredictable to the verifier. To verify it, the verifier must be able to reset data memory into a known or predictable state before attestation. The verification function can reset data memory into a known state by erasing the contents of data memory. To prevent attacker from predicting or compressing the contents of data memory, the data memory should be filled with pseudo-random values before attestation.

3 Problem Definition, Assumptions and Attacker Model

Problem Definition. We consider the problem of how a host machine can verify the firmware integrity of a peripheral attached to it without any dedicated hardware, i.e., using a software-only approach that can detect arbitrary malicious changes.

Assumptions. We assume that the verifier knows the exact hardware configuration of the peripheral, such as the CPU model, the CPU clock speed, the size of

program memory, and the size of data memory. We also assume that the verifier has a copy of the expected contents of the program memory of the peripheral. We assume that the communication channel between the peripheral and the verifier can provide message-origin authentication. We also assume that the peripheral can only communicate with the verifier during the attestation, which prevents the peripheral from communicating with a faster machine to compute the checksum (this attack is called a proxy attack). This can be implemented through a physical connection, such as USB cable.

Attacker Model. We assume that an attacker cannot change the hardware configuration of peripherals, such as speeding up the CPU clock, or adding additional program memory or data memory. However, the attacker can make arbitrary changes to the peripheral software. We assume that there are software vulnerabilities in the peripheral firmware, through which an attacker can attempt a compression or ROP attack.

4 SBAP: Software-Based Attestation for Peripherals

4.1 System Overview

Similar to previous proposals [2, 3], SBAP verifies the firmware integrity of a peripheral through a challenge-response protocol and a predicted computation time constraint. To prevent known attacks, SBAP leaves no available empty space in memory for attackers by filling all unused space in program and data memory with pseudo-random values, and verifying the integrity of both program and data memory. Also, different from SWATT and ICE, SBAP utilizes an efficient pseudo-random number generator, which is mainly designed for low-speed devices with constrained computation and memory resources. Figure 1 depicts an overview of the system setup as well as the protocol steps. On the peripheral, the verification function is responsible for disabling all interrupts in the microcontroller, filling the data memory with pseudo-random values, computing the checksum over the entire contents of data memory and program memory, and sending the final checksum result to the verifier. Before peripheral deployment, available free space in program memory must be filled with pseudo-random values. On the verifier, a nonce generator generates random nonces to seed the untrusted peripheral and a timer measures the verification time. A checksum simulator on the verifier computes the expected checksum result by simulating the checksum procedure.

4.2 Protocol

To verify the firmware integrity on a peripheral, SBAP performs the following steps:

1. The verifier sends the attestation request to the untrusted peripheral. An 8-byte random nonce is included in this request as a seed to the verification function.

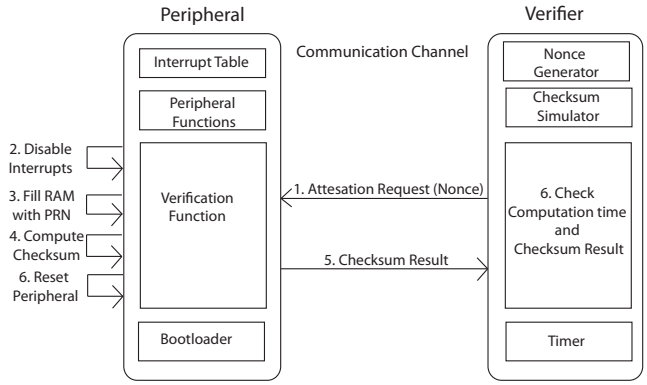

Fig. 1. System Overview and Protocol

2. Upon receiving the attestation request, the verification function first disables all interrupts on the peripheral to set up an untampered execution environment.
3. The verification function fills the entire data memory with pseudo-random values generated using the nonce as a seed.
4. The verification function computes a checksum over the entire contents of both program memory and data memory. Our checksum function uses part of data memory to store variables. The values of these variables are predictable to the verifier though they change dynamically during checksum computation.
5. The verification function sends the checksum result to the verifier.
6. The verifier verifies the checksum result as well as the computation time. If the verifier receives the expected checksum result within predicted time, the verifier trusts the attached peripheral. Otherwise, the verifier rejects the attached peripheral. At the same time, the peripheral is reset so that the pseudo-random values in data memory are cleaned, and the registers of the peripheral are restored to their default states.

After the attestation, since data memory is filled with pseudo-random values, the peripheral should be reset before being used. Additionally, during the reset, some of the registers should be restored to their default states. Otherwise, an attacker may get a chance to hide malicious data in the register space for future attacks. An attacker may store the malicious data in some IO data or control registers that are never used by the peripheral. For example, on a CY7C63923 microcontroller, there are 17 bytes of IO data registers that an attacker can use to hide malicious data for future attacks. In general, a power on reset is always enabled on a peripheral, during which all the registers are restored to their default values. A software-based reset function is also a solution if the function can prevent an attacker from hiding any malicious data in register space.

4.3 Verification Function

Due to the constrained computation and memory resources in the simple microcontrollers that are deployed on low-speed peripherals, the verification functions that are used in previous proposals cannot be deployed directly in SBAP. In this section, we detail the verification design in SBAP by describing the pseudo-random number generator, our design for filling data memory with pseudo-random values, and our checksum function.

Pseudo-Random Number Generator (PRNG). In the verification function, the PRNG is used for two purposes:

1. output PRNs to fill the data memory;
2. output PRNs to construct memory address to read in a pseudo-random fashion.

In previous proposals [2], T-functions [11] or RC4 [12] are used to output PRNs. However, on low speed peripherals, it is challenging to implement the same PRNGs efficiently due to constrained computation or memory resources. T-functions need a multiplication unit to generate PRNs efficiently. However, a hardware multiplication unit is not available in many low-speed microcontrollers that are used in peripherals. Software-based multiplication is too slow to be a viable option. For instance, on a CY7C63923 microcontroller [6], a software-based multiplication requires thousands of cycles to complete a 16-bit multiplication. An RC4-based PRNG outputs pseudo-random numbers through simple arithmetic and logical operations. However, RC4 requires at least 256 bytes of RAM, which consumes all memory resource on some microcontrollers (such as the CY7C63923). Laszlo et al. [13] propose several efficient PRNGs that are primarily designed for low speed embedded devices. The PRNGs proposed by Laszlo et al. only require simple addition, XOR, or shift operations and few memory resources to output PRNs efficiently. From the PRNGs that Laszlo et al. propose, we select a 2-stage PRNG in our SBAP design. Other PRNGs that have the same features are also potential choices for SBAP. The PRNG we select outputs PRNs as follows:

$$x[i+1] = x[i-1] + (x[i] \oplus rot(x[i-1], 1)) \tag{1}$$

$\oplus$ is the logical XOR operation and *rot* is the left rotation shift operation. x is the output of this PRNG, a 32-bit long stage. The value of one stage is updated based on the values of the previous two stages in each iteration.

Filling Data Memory With Pseudo-Random Values. The verification function fills data memory in a pseudo-random fashion. Such a design is required to prevent an attacker from reserving one small block at the end of data memory to store malicious data, and then generating the PRNs that are expected to be in that small block of data memory on-the-fly when they are needed by the checksum routine. In our design, the verification function determines the data

memory addresses to be filled based on the outputs of the PRNG. Each address is then filled using the XOR of two bytes of PRNG output. This prevents the attacker from generating the PRNs that are expected in data memory based on the values of existing PRNs in other locations in data memory, since only XOR results are stored in data memory. To make sure that all data memory is filled, the verifier can obtain the number of loop iterations upon which all data memory has been filled. This value is determined by simulating the filling procedure before sending the attestation request.

Checksum Function Design. The checksum function computes a fingerprint over the entire contents of both program memory and data memory. As in SWATT or ICE, the checksum is computed through a strongly ordered sequence of addition, XOR, and rotation shift operations. If the sequence of the operations is altered or some operations are removed, the checksum result will be different with a high probability. Also, the checksum function reads memory in a pseudo-random traversal. If the memory size is N bytes, each memory location is accessed at least once after $O(NlnN)$ memory read with a high probability [2]. The input to the checksum function is a 16-byte pseudo-random value, which is used to seed the PRNG (i.e., to provide stages $x[0]$ and $x[1]$) and to initialize an 8-byte checksum vector. The output of the checksum function is also an 8-byte checksum vector. Each byte of the checksum vector is called a checksum state. For each iteration of the checksum function, the value of one checksum state is updated based on the current memory contents, the pseudo-random value, and the values of other checksum states. In order to preserve the entropy of the carry bit, we add each carry bit to the checksum state. Following is the pseudo code of one iteration of the checksum function:

```
/* C is the checksum vector, i is its current index. */
/* PRN is the pseudo-random number */
/* addr is the memory address */
   addr = PRN & MASK     /* Construct memory address */
/* update one checksum state */
   C[i] = C[i] + ( Mem[addr] xor C[(i-2) mod 8] )
   tmp = C[i] mod 256
   C[i] = rotation_left_shift(tmp, 1) + ( C[i] >> 8 )
   C[i] = C[i] + i
   i = (i + 1) mod 8           /* update the index i */
```

To optimize the computation time of the checksum, we unroll the checksum loop eight times and each time one checksum state is updated by either the contents of program memory or the contents of data memory, which can be adjusted based on the memory size proportion of each. For example, on a peripheral that has 8 KB of programmable Flash and 256-bytes of RAM, seven checksum states can be updated based on the contents of Flash memory while one checksum state can be updated based on the contents of RAM. To make the computation

of each checksum state different, we also add the index value to the checksum state.

5 Implementation

In this section, we detail the implementation of SBAP on a wired Apple Aluminum Keyboard.

5.1 The Apple Aluminum Keyboard

The Apple Aluminum Keyboard connects to a computer via a USB interface. Inside the Apple Aluminum keyboard, a Cypress CY7C63923 microcontroller controls the keyboard matrix. During a firmware update, the firmware on the CY7C63923 microcontroller is updated. The Cypress CY7C63923 microcontroller belongs to the Cypress enCoReTM II family and is primarily designed for low-speed USB peripheral controllers, such as mice, keyboards, joysticks, game pads, barcode scanners, and remote controllers. The Cypress CY7C63923 is a Harvard Architecture, 8-bit programmable microcontroller with 256 bytes of RAM and 8 KB of Flash. Five registers on this microcontroller control the operations of its CPU. These five registers are the Flag register (F), Program Counter (PC), Accumulator Register (A), Stack Pointer (SP), and Index Register (X). PC is 16-bits in length, while all the other registers are 8-bits long. A and I are used during arithmetic or logical operations on this microcontroller.

5.2 Verification Function

Following a keyboard firmware update, the Flash memory from 0xe00 to 0x1300 (1280 bytes) is available free space, where we implement our verification function. Figure 2 shows the final memory layout of keyboard Flash memory. The verification function is located at addresses 0x0e00 – 0x1268 in the Flash memory. The Flash memory from 0x1268 to 0x1300 is filled with pseudo-random values. In the verification function, a 'Send Function' is the communication module that handles the attestation request from the verifier and returns checksum results through the USB channel to the verifier following checksum computation. Before the attestation, the contents of RAM is unpredictable to the verifier. Therefore, an 'Initial Function' sets the contents of data memory to a known state by filling the data memory with pseudo-random values (we fill data memory in a linear sequence instead of in a pseudo-random fashion as designed). The data memory from 0x18 to 0xff is filled with with pseudo-random values, while the data memory from 0x00 to 0x17 is used to store variables for the verification function. Also, the 'Initial Function' disables all interrupts on the CY7C63923 microcontroller, which prevents the contents of data memory from being modified by an interrupt call during checksum computation. A 'Checksum Function' is implemented, which computes a checksum over the entire contents of both program memory (Flash) and data memory (RAM). After attestation, we reset the Apple

Aluminum Keyboard. A two-stage pseudo-random number generator (PRNG) is implemented in both the 'Initial Function' and 'Checksum Function'. The 8-byte nonce sent by the verifier is used to seed the PRNG in the 'Initial Function'. After filling RAM, the PRNG in 'Initial Function' outputs a 16-byte random number to serve as input to the 'Checksum Function', which is used to seed the PRNG in 'Checksum Function' and to initialize the 8-byte checksum vector. All of these functions are implemented in assembly. The two-stage PRNG is implemented using 23 assembly instructions. It outputs 4 bytes of pseudo-random values every 157 CPU cycles on the Apple Aluminum Keyboard. We unroll the checksum iteration eight times. Each time one checksum state is updated. The first seven checksum states are updated based on the content of Flash memory while the last checksum state is updated based on the content of RAM. Including the two-stage PRNG, 'Checksum Function' only requires 19.5 instructions and 133.5 CPU cycles on the average to update one checksum state on the Apple Aluminum Keyboard. Following is the assembly we implement to update one checksum state:

```
; [0x00] to [0x07] saves outputs of PRNG
; [0x08] to [0x0f] saves temp variables, such as counter
; [0x10] to [0x17] saves checksum states
; ROMX is the instruction to read flash memory
; CPU loads memory address from register A
; and register X when ROMX is executed
; the result of ROMX is saved in register A automatically by CPU
; Update checksum[0]
MOV X, [0x00]     ; read pseudo-random values
MOV A, [0X01]     ; to register X and A
AND A,0X1F        ; construct memory address
ROMX              ; read Flash memory, result is saved in A
XOR A, [0x16]     ; Mem[addr] xor checksum[6]
ADD [0x10], A     ; add previous checksum value
RLC [0x10]        ; left rotation shift 1 bit, add carry bit
ADC [0x10],0x00   ; add carry bit (add index too)
```

6 Experimental Results

Verification Time. Figure 3 shows the verification time for 40 trials. In each trial, the verifier measures the entire verification time between sending a nonce to the Apple Aluminum Keyboard and receiving the checksum result from the keyboard. The average verification time of the 40 trials is 1706.77 ms while the standard deviation is only 0.18 ms.

USB Communication Overhead. In this experiment, the verifier first sends an attestation request to the Apple Aluminum Keyboard. Upon receiving a request from the verifier, the verification function on the Apple Aluminum Keyboard

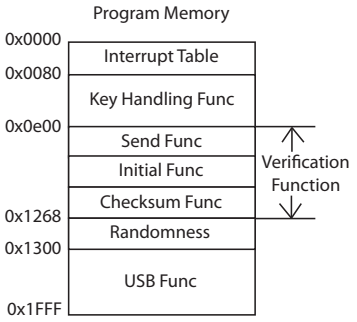

Fig. 2. Memory layout of program memory

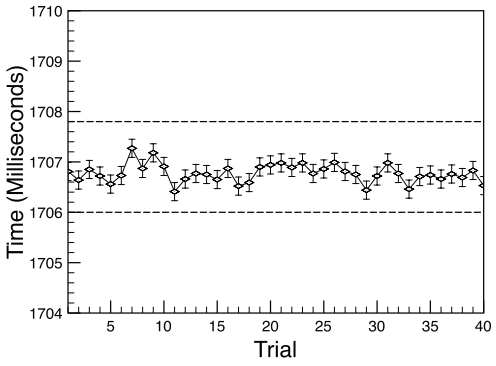

Fig. 3. Verification Time

returns an 8-byte value to the verifier immediately without computing the checksum. To obtain accurate experimental results, the verifier measures the entire time of 1000 runs of the communication in each trial. Figure 4 shows the average communication time of the 1000 runs in each trial. The average value of the USB communication overhead for all the experiments is 1.83 ms and the standard deviation is only 0.01 ms.

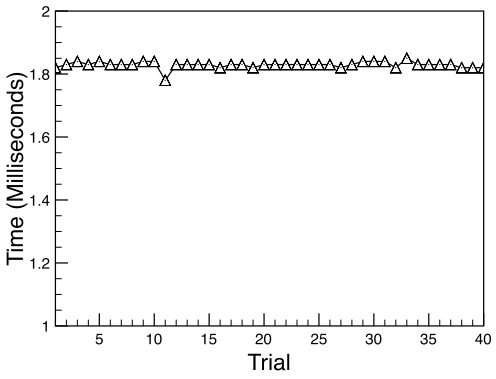

Fig. 4. USB Communication Overhead

Analysis. The experimental results show that the verification procedure is very stable. As shown in Figure 3, the verification time for all 40 trials varies from about 1706 ms to about 1708 ms. An attacker cannot hide malicious code from an attestation unless the malicious code computes the correct checksum result with a computation overhead less than 3 ms, which is only about 0.2 percent of the verification time. This kind of attack is extremely challenging for the attacker since there is not any free space left in program or data memory. Also, the experimental results show that the communication overhead does not affect the detection of the computation overhead caused by malicious code, since the communication is also very efficient and stable. Another important result we obtain from the system evaluation is that SBAP is an efficient and realistic solution to solve the peripheral integrity verification problem. In our prototype, a verifier (a user) only needs to wait about 2 seconds for the entire verification procedure, which is acceptable if run in response to being connected to a host.

7 Discussion

To the best of our knowledge, there is no efficient attack against SBAP. One possible attack is that an attacker performs a compression attack or ROP attack and stores the malicious data or code in data memory, then generates the pseudo-random values that should be filled in data memory on-the-fly during

the checksum computation. However, this attack causes large computational overhead due to the complexity of the PRNG. Another attack is that an attacker performs a memory copy attack by having a copy of the original code or pseudo-random values that should be in data memory in the register space of the microcontroller. This attack can be detected by the verifier since a memory copy attack causes a detectable computation overhead. In fact, there is very low likelihood that an attacker can perform this attack because there is not much register space available for an attacker to perform a memory copy attack on a peripheral. For example, we find that on a CY7C36923 microcontroller there are about 30 registers (30 bytes) that can be used for a memory copy attack by an attacker. However, once the attacker changes the checksum loop, the attacker needs hundreds of bytes memory to store the original copy of the checksum function. Finally, an attacker can hide some malicious data that can be used for future attacks in the register space of a peripheral. SBAP prevents this attack by resetting the peripheral after an attestation.

8 Related Work

Several software based attestation technologies on embedded devices have been proposed. Seshadri et al. propose SWATT [2], ICE, SCUBA [3], and SAKE [4], as discussed in Section 2. However, as discussed in this paper, it is challenging to implement ICE in resource-constrained embedded devices. To prevent adversary from hiding malicious code in the empty memory, Yang et al. suggest filling the available empty space with pseudo-random values [14]. Castelluccia et al. discuss the difficulties of software-based attestation on embedded devices [5]. While they mention many important points, we pointed out several errors and inaccuracies [15].

9 Conclusions and Future Work

We propose SBAP, a software-only solution to verify the firmware integrity of peripherals. SBAP verifies the contents of both program memory and data memory in a peripheral and detects malicious changes with high probability in the face of recently proposed attacks (e.g., a memory copy or memory substitution, a compression attack, and a ROP attack). We implement and evaluate SBAP on an Apple Aluminum Keyboard. One area of future work is to implement and evaluate SBAP on other peripherals, especially high-speed peripherals such as a network interface. The hardware architecture and configuration of a high-speed peripheral is different from those of a low speed peripheral. Therefore, there will likely be new challenges when we evaluate SBAP on a high-speed peripheral.

References

1. Chen, K.: Reversing and exploiting and Apple firmware update. In: Black Hat. (2009)

2. Seshadri, A., Perrig, A., van Doorn, L., Khosla, P.: SWATT: Software-based attestation for embedded devices. In: Proceedings of the IEEE Symposium on Security and Privacy. (2004)
3. Seshadri, A., Luk, M., Perrig, A., van Doorn, L., Khosla, P.: SCUBA: Secure code update by attestation in sensor networks. In: ACM Workshop on Wireless Security (WiSe 2006). (2006)
4. Seshadri, A., Luk, M., Perrig, A., Van Doorn, L., Khosla, P.: SAKE: Software attestation for key establishment in sensor networks. In: International Conference on Distributed Computing in Sensor Systems. (2008)
5. Castelluccia, C., Francillon, A., Perito, D., Soriente, C.: On the difficulty of software-based attestation of embedded devices. In: Proceedings of ACM Conference on Computer and Communications Security (CCS). (2009)
6. CYPRESS: CYPRESS enCoRe II low-speed USB peripheral controller (CY7C639XX)
7. Huffman, D.: A method for the construction of minimum redundancy codes. In: Proceedings of the IRE 40. (1962)
8. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: Generalizing return oriented programming to RISC. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS). (2008)
9. Hund, R., Holz, T., C, F.F.: Return oriented rootkit: Bypassing kernel code integrity protection mechanisms. In: Proceedings of the 18th USENIX Security Symposium. (2009)
10. Shacham, H.: The geometry of innocent flesh on the bone: Return into libc without function calls (on the x86). In: Proceedings of the ACM Conference on Computer and Communications Security (CCS). (2007)
11. Klimov, A., Shamir, A.: A new class of invertible mappings. In: CHES 02: Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems. (2003) 470 to 483
12. wikipedia: http://en.wikipedia.org/wiki/RC4
13. Hars, L., Petruska, G.: Pseudo-random recursions: Small and fast pseudo-random number generator for embedded applications. In: EURASIP Journal on Embedded Systems. (2007)
14. Yang, Y., Wang, X., Zhu, S., Cao, G.: Distributed software-based attestation for node compromise detection in sensor networks. In: Proceedings of IEEE International Symposium on Reliable Distributed Systems. (2007)
15. Perrig, A.: Refutation of "on the difficulty of software-based attestation of embedded devices". `http://sparrow.ece.cmu.edu/group/pub/ccs-refutation.pdf` (2010)