Practical Analysis Framework for Software-based Attestation Scheme

Li Li[1], Hong Hu[1], Jun Sun[2], Yang Liu[3], and Jin Song Dong[1]

[1] National University of Singapore
[2] Singapore University of Technology and Design
[3] Nanyang Technological University

Abstract. An increasing number of "smart" embedded devices are employed in our living environment nowadays. Unlike traditional computer systems, these devices are often physically accessible to the attackers. It is therefore almost impossible to guarantee that they are un-compromised, i.e., that indeed the devices are executing the intended software. In such a context, software-based attestation is deemed as a promising solution to validate their software integrity. It guarantees that the software running on the embedded devices are un-compromised without any hardware support. However, designing software-based attestation protocols are shown to be error-prone. In this work, we develop a framework for design and analysis of software-based attestation protocols. We first propose a generic attestation scheme that captures most existing software-based attestation protocols. After formalizing the security criteria for the generic scheme, we apply our analysis framework to several well-known software-based attestation protocols and report various potential vulnerabilities. To the best of our knowledge, this is the first practical analysis framework for software-based attestation protocols.

1 Introduction

"Smart" sensory embedded devices are getting more and more popular nowadays. They are frequently used for temperature measurement, fire detection, water saving, etc. In the near future, they are expected to be ubiquitous. However, their wide adoption poses threats to our safety and privacy as well. Unlike traditional computer systems, these devices are often physically accessible to the attackers and it is almost impossible to guarantee that they are un-compromised, i.e., that indeed the devices are executing the intended software. Effective techniques for verifying and validating the embedded devices against malicious adversary becomes increasingly important and urgent. Traditional hardware-based attestation [1–4] is cost-ineffective in such a context. Thus, software-based attestation [5–7], which aims to function without any dedicated security hardware, is deemed as a promising solution for verifying the integrity of these massive, inexpensive, and resource constrained devices.

Software-based attestation is based on the challenge-response paradigm between the trusted verifier and the potentially compromised prover (the embedded device). It typically works as follows. The verifier first sends a random challenge to the prover and asks the prover to generate a checksum for its memory state based on the challenge. Since the prover's computing and memory resources are designed to be fully utilized

in the attestation, if the memory is tampered by the adversary, the prover needs to take extra time to compute the correct checksum. We further assume that the verifier knows the expected memory state of the prover. He thus can compute the same checksum and compare it with the one received from the prover. By exploiting the fact that the prover is resource constrained, software-based attestation ensures that the prover can return the correct response in time only if it is genuine. On the other hand, whenever the prover fails to reply in time or returns an incorrect checksum, it is highly likely compromised.

The software-based attestation protocol design is challenging and error-prone [8, 9]. Hence, in this work, we propose an analysis framework for software-based attestation that can be easily adopted in practice. First, our framework provides a parameterized generic software-based attestation scheme that captures most existing software-based attestation protocols. The adversary modeled in this work can not only compromise the prover before the attestation, but also communicate with the compromised prover during the attestation. We then formalize the security criteria for the generic scheme based on the knowledge of network latency (which is important as timing is essential here) and adversary model. Since the real software-based attestation protocols are instances of the generic scheme, these criteria thus naturally should be hold in the real protocols as well. Hence, we apply our analysis framework to three well-known software-based attestation schemes, i.e., SWATT [5], SCUBA [7] and VIPER [10], and find four potential vulnerabilities that have not been reported before. As far as we know, this is the first framework that can give practical analysis to real software-based attestation protocols.

2 Generic Specification for Software-based Attestation

We start with defining a generic software-based attestation scheme which captures most existing software-based attestation protocols. The idea is that analysis results based on the generic schema can be extended to concrete protocols readily as we show in later sections. The generic software-based attestation scheme involves three parties, i.e., the trusted verifier $\mathcal{V}$, the prover (the embedded device) $\mathcal{P}$ and the adversary $\mathcal{A}$. We denote the genuine prover and the compromised prover as $\mathcal{P}_g$ and $\mathcal{P}_c$ respectively. In this section, we first present the system model, including the system architecture, the security property and the threat model. Then we propose a generic software-based attestation scheme between the trusted verifier $\mathcal{V}$ and the genuine prover $\mathcal{P}_g$ based on our system.

2.1 System Overview

Software-based attestation is proposed to verify the resource constrained embedded devices without using any security hardware (e.g., TPMs [11]). Before presenting the details of the generic attestation scheme, we first describe the system model employed in this work. The attestation procedure is conducted between a trusted verifier $\mathcal{V}$ and a prover $\mathcal{P}$ over the network. We explicitly consider the network round-trip time (RTT).

The architecture of the verifier $\mathcal{V}$ and the prover $\mathcal{P}$ considered in this work are depicted as follows. $\mathcal{P}$ consists of a computing processor, several registers and a memory M. The data memory M_d and the program memory M_p are two different memory

space that should be attested in M. Specifically, M_d stores the runtime data (e.g., stack information, data collected from the environment) that are unpredictable to $\mathcal{V}$, hence its content cannot be attested directly in the attestation procedure. M_p stores the program code which is known to $\mathcal{V}$. The attestation routine $verif$ on the prover side is pre-installed in M_p before the attestation starts. In general, the size of M_d could be 0 when the attestation for the data memory is not required. Notice that some memory can be excluded from the attestation in some specific attestation protocols [12, 7, 10], and thus $M_d + M_p$ may not equal to M. Meanwhile, $\mathcal{V}$ is a powerful base station who can simulate the execution of $\mathcal{P}$. When $\mathcal{V}$ has the image of both M_d and M_p in $\mathcal{P}$, $\mathcal{V}$ can compute the memory checksum based on the image.

During the attestation, $\mathcal{P}$'s data memory M_d will be first overwritten into a state that is known to $\mathcal{V}$. The attestation then aims at verifying whether $\mathcal{P}$ has a genuine state for both M_d and M_p as $\mathcal{V}$ expected. Let $State(\mathcal{P})$ be the memory state of $M_d + M_p$ in the prover $\mathcal{P}$. When $State(\mathcal{P})$ is known to $\mathcal{V}$, the attestation can be modeled by a game between the verifier $\mathcal{V}$ and the prover $\mathcal{P}$. In the game, $\mathcal{V}$ first sends a random challenge to $\mathcal{P}$, and then $\mathcal{P}$ picks a checksum reply based on the challenge. The prover $\mathcal{P}$ wins if the used time is less than some threshold and the checksum is correct, otherwise $\mathcal{P}$ loses the game. We denote the percentage of differences between two memory states S and S' as $\lambda(S, S')$ and the winning probability of $\mathcal{P}$ as $\mathbb{P}_w(\mathcal{L}, \mathcal{P})$, where $\mathcal{L}$ denotes the system and its configurations. We define an attestation protocol as *correct* if $\mathbb{P}_w(\mathcal{L}, \mathcal{P}_g) = 1$, which means that the genuine prover $\mathcal{P}_g$ can always win. On the other hand, when μ is the least memory proportion that should be modified in the compromised prover $\mathcal{P}_c$ to perform a meaningful attack, we define an attestation protocol as $\langle \varepsilon, \mu \rangle$-*secure* if $\forall \mathcal{P}_c, \lambda(State(\mathcal{P}_c), State(\mathcal{P}_g)) \geq \mu > 0 \Rightarrow \mathbb{P}_w(\mathcal{L}, \mathcal{P}_c) \leq \varepsilon$, which means that any prover who needs to overwrite at least μ percentage of the attested memory has the winning probability of no more than ε. In the attestation, the adversary wins if and only if he can keep the malicious code in the attested memory after the attestation. However, software-based attestation does not guarantee that the device is unmodified before the attestation.

The adversary $\mathcal{A}$'s capability is specified with two phases. Before the attestation begins, $\mathcal{A}$ can use unlimited resources to reprogram the memory in $\mathcal{P}_c$. However, $\mathcal{A}$ cannot change the physical hardware and the network infrastructure, so $\mathcal{P}_c$'s memory storage, computing power and network latency are fixed. Once the attestation starts, $\mathcal{A}$ cannot modify $\mathcal{P}_c$'s memory content anymore. Nevertheless, $\mathcal{A}$ can communicate with $\mathcal{P}_c$ over the network and compute with unlimited resources.

Notations. The notations used in this paper are listed as follows. We write X, Y, Z to denote sets and x, y, z to denote elements in the sets. $f(x : X, y : Y) \rightarrow z : Z$ represents a function f that maps the tuple of two elements x, y to the element z. Let n be a natural number. X^n stands for the concatenation of n elements in X. $X \times Y$ is the Cartesian product of X and Y. Let $\mathbb{D}$ be a probabilistic distribution over set X. $x \leftarrow\!\!\!\!\shortmid\ \mathbb{D}\ \shortmid\!\!\!\!- X$ means assigning an element of X to x according to $\mathbb{D}$. $[n \dots m]$ represents the integers from n to m. $[n, m]$ stands for the real numbers from n to m. $max_{x,y}\{f(x, y)\}$ stands for the maximum value of $f(x, y)$ for any x and y. $Pr[x]$ denotes the probability of x.

Checksum Computation $comp(S_a, g_0, r_0)$
S_a is the memory state of $\mathcal{P}$ under attestation.
g_0 is the address generator seed.
r_0 is the checksum response seed.

for i **in** $[1 \dots n]$ **do**
  $g_i = Gen(g_{i-1})$;
  $a_i = Addr(g_i)$;
  $c_i = Read(S_a, a_i)$;
  $r_i = Chk(r_{i-1}, c_i)$;
end
return r_n;

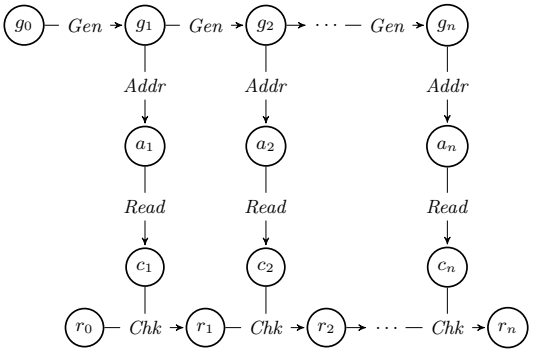

Fig. 1: Checksum Computation

2.2 Generic Attestation Scheme

In this section, we propose a generic specification for software-based attestation scheme that captures most existing software-based attestation protocols. The specification is described in two parts. First, given a memory state $S_a = State(\mathcal{P})$ of both M_d and M_p, we introduce the *checksum computation routine* that compute the memory checksum as shown in Figure 1. Then, we illustrate the *generic software-based attestation scheme* which first securely erases the data memory M_d and then attests the whole memory $M_d + M_p$ with the *checksum computation routine*.

The checksum computation routine $comp(S_a, g_0, r_0)$ aims at computing the unforgeable checksum for memory state S_a based on the initial address generator g_0 and initial memory checksum r_0. It iteratively computes the address generator g_i, the memory address a_i, the memory content c_i and the checksum response r_i for $i \in [1 \dots n]$ as shown in Figure 1. The four functions used in the generic scheme are illustrated as follows. In the following paper, l_g, l_a, l_c and l_r represent lengths of g_i, a_i, c_i and r_i respectively.

- $Gen(g_{i-1} : \{0,1\}^{l_g}) \rightarrow g_i : \{0,1\}^{l_g}$ computes the generator g_i of the memory addresses in a random manner incrementally.
- $Addr(g_i : \{0,1\}^{l_g}) \rightarrow a_i : \{0,1\}^{l_a}$ converts the random generator g_i to the memory address a_i.
- $Read(S_a : \{0,1\}^{l_a} \times \{0,1\}^{l_c}, a_i : \{0,1\}^{l_a}) \rightarrow c_i : \{0,1\}^{l_c}$ reads the memory content c_i located at the address a_i in S_a.
- $Chk(r_{i-1} : \{0,1\}^{l_r}, c_i : \{0,1\}^{l_c}) \rightarrow r_i\{0,1\}^{l_r}$ updates the last checksum response r_{i-1} with the memory content c_i to the new checksum r_i.

The generic software-based attestation scheme is shown in Figure 2. The functions used in the figure are illustrated as follows. $rand(x)$ generates a random bit-string and stores it into x. $fill(M, S)$ fills the memory M with state S. $Gen_0(o, g_0)$ and $Chk_0(o, r_0)$ derive the initial values for the generator and the checksum from the challenge o and store them into g_0 and r_0 respectively. $comp(S_a, g_0, r_0)$ illustrated previously computes the checksum for memory state S_a with the generator seed g_0 and

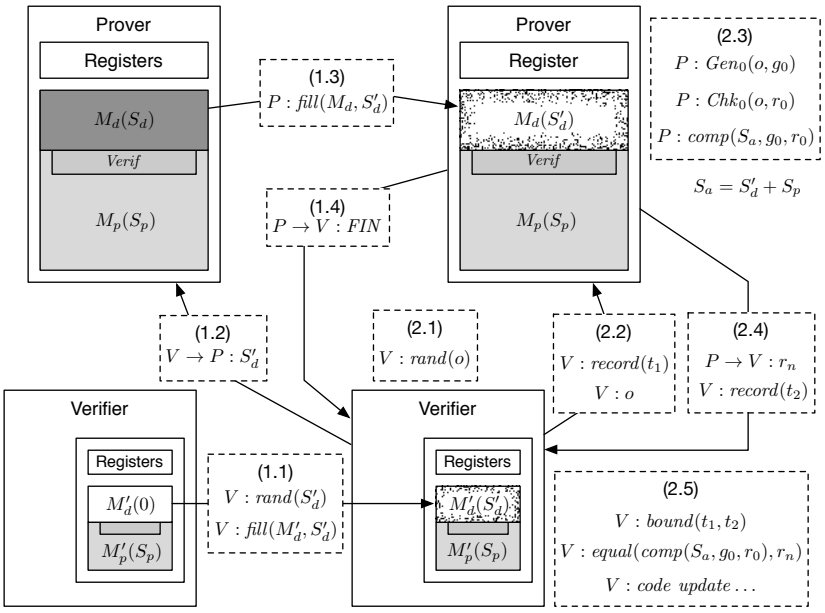

Fig. 2: Generic Software-based Attestation Scheme

the response seed r_0. $record(t)$ records the current time into t. $bound(t_1, t_2)$ checks whether $t_2 - t_1$ is smaller than a time bound. $equal(x, y)$ checks if x and y are equivalent. $I : op$ means that I conducts the operation op. $I_1 \rightarrow I_2 : m$ means that I_1 sends the message m to I_2. The generic software-based attestation scheme proposed in this work is divided into two phases as shown in Figure 2.

Phase 1. Secure Erasure overwrites the data memory M_d with random noise. Initially, $\mathcal{P}$'s data memory image M'_d in $\mathcal{V}$ are filled with 0, while M_d in $\mathcal{P}$ has the memory state S_d consisting of information generated at runtime. At the end of this phase, $\mathcal{P}$ and $\mathcal{P}$'s image in $\mathcal{V}$ have the same memory state S'_d filled with random noise.

1. When $\mathcal{V}$ wants to start the attestation, it first overwrites $\mathcal{P}$'s data memory image M'_d in $\mathcal{V}$ to a random state S'_d, which is generated by the $rand(S'_d)$ function.
2. $\mathcal{V}$ sends S'_d to $\mathcal{P}$ and asks $\mathcal{P}$ to overwrite its M_d with S'_d.
3. $\mathcal{P}$ accepts $\mathcal{V}$'s requests and updates his M_d with S'_d. In fact, the last step (1.2) and this step (1.3) can be streamlined. Whenever $\mathcal{P}$ receives a value from $\mathcal{V}$, he writes it into the corresponding data memory location.
4. When M_d is filled with S'_d, $\mathcal{P}$ sends a FIN signal to start the second phase.

Phase 2. Checksum Computation aims at attesting both M_d and M_p in $\mathcal{P}$ and discovering memory modification with overwhelming probability. When the first phase is finished, $\mathcal{V}$ can run the second phase for multiple times consecutively. Upon the beginning of the second phase, $\mathcal{V}$ knows the memory state $S_a = State(\mathcal{P})$.

1. $\mathcal{V}$ first picks a random challenge o.

2. $\mathcal{V}$ sends o to $\mathcal{P}$ and asks $\mathcal{P}$ to compute the checksum for his memory state $S_a = S_p + S'_d$. $\mathcal{V}$ also records the time t_1 when the request is sent.
3. After $\mathcal{P}$ derives the initial address generator g_0 and the initial checksum response r_0 from the challenge o, he computes the checksum over the memory state S_a with $comp(S_a, g_0, r_0)$ illustrated in Figure 1.
4. As soon as the checksum computation routine is finished, $\mathcal{P}$ sends the checksum r_n back to $\mathcal{V}$. $\mathcal{V}$ again records the time t_2 when r_n is received.
5. Once $\mathcal{V}$ receives r_n from $\mathcal{P}$, he checks two conditions: (1) whether the checksum is received within the timing threshold $\{bound(t_1, t_2) = true\}$ and (2) whether the checksum is correct $\{equal(comp(S_a, g_0, r_0), r_n) = true\}$. If both of the conditions are satisfied, $\mathcal{P}$ is trusted as genuine and $\mathcal{V}$ will update $\mathcal{P}$'s unattested memory. Otherwise, $\mathcal{P}$ is deemed as compromised.

Assumptions. In order to guarantee the correctness of the protocol, we make the following assumptions. First, $\mathcal{P}$ either has the attestation procedure $verif$ pre-deployed in its program memory M_p or can download it into a pre-allocated memory space in M_p at runtime before the attestation starts. Second, $\mathcal{V}$ knows the exact memory image of M_p in $\mathcal{P}$. M_d and M_p share the same address space. Third, the attestation procedure $verif$ implemented in $\mathcal{P}$ is optimal in terms of execution speed. Fourth, S'_d and o are unpredictable to the prover. Fifth, the cryptographic primitives used in the attestation procedure are perfect. This assumption does not reduce the security offered by our framework to the real applications. We can update the attestation procedure with the state-of-the-art cryptographic implementations that are unbreakable at the moment. For instance, when a hash function is needed in the attestation, we use *SHA-2* or *SHA-3* that are safe for the time being. Sixth, the adversary cannot personate the prover and communicate with the verifier directly, which means that the verifier is connected to the prover via a controllable channel during the attestation, e.g., a *bus* used in [10]. When the adversary can personate the prover, the software-based attestation protocol is trivially broken because the adversary can answer the challenge for the prover.

3 Security Criteria Formalization

In this section, we introduce several attack scenarios. Based on the attacks, we formalize the security criteria for the generic attestation scheme. When the compromised prover $\mathcal{P}_c$ computes the checksum by itself, we need to discuss two cases: (1) the checksum is computed with the checksum computation routine at runtime, or (2) the checksum is pre-computed. In the first case, when the memory and the registers are fully utilized as shown in Section 3.1, we measure the winning probability of $\mathcal{P}_c$ who trades computation power for memory space (*memory recovering attack*) in Section 3.2. In the second case, we discuss the scenario where $\mathcal{P}_c$ stores the pre-computed challenge-response pairs in the its memory (*challenge buffering attack*) in Section 3.3. On the other hand, when $\mathcal{P}_c$ does not compute the checksum by itself, it can ask $\mathcal{A}$ to compute the checksum (*proxy attack*) as introduced in Section 3.4. When the memory and the registers are fully attested, since the above three attack methods are orthogonal, the winning probability of the compromised prover $\mathbb{P}_w(\mathcal{L}, \mathcal{P}_c)$ then can be calculated by the most effective attack among them. Some used notations are summarized in Table 1.

Name	Explanation	Size
$M_d(S_d)$	Data memory M_d filled with memory image state S_d	m_d unit
$M_p(S_p)$	Program memory M_p filled with memory image state S_p	m_p unit
$M(S)$	Overall memory M filled with memory image state S	m unit [a]
o	The challenge sent from $\mathcal{V}$ to $\mathcal{P}$	l_o bit
g_i	Address generators for $i \in [0 \dots n]$	l_g bit
a_i	Memory addresses for $i \in [0 \dots n]$	l_a bit
c_i	Memory contents for $i \in [0 \dots n]$	l_c bit
r_i	Checksum responses for $i \in [0 \dots n]$	l_r bit
$T_{\mathcal{V}}^{min}, T_{\mathcal{V}}^{max}$	Network RTT between $\mathcal{V}$ and $\mathcal{P}_g$ varies from d_g^{min} to d_g^{max}	-
$T_{\mathcal{A}}^{min}, T_{\mathcal{A}}^{max}$	Network RTT between $\mathcal{A}$ and $\mathcal{P}_c$ varies from d_c^{min} to d_c^{max}	-
$d_{Gen}, d_{Addr}, d_{Read}, d_{Chk}$	Computation time for Gen, $Addr$, $Read$ and Chk resp.	-
d_g	The time needed by $\mathcal{P}_g$ to compute the memory checksum	-
d_{th}	The timing threshold on the verifier side	-
n	The number of iterations in a single checksum computation	-
k	The number of consecutive checksum computation (Phase 2)	-
u	The number of registers used to store the checksum	-

Table 1: Notation Summary

[a] m may not equal to $m_d + m_p$ when some memory is left unattested.

3.1 Full Utilization of Memory and Registers

In the checksum computation routine, the memory are accessed in a random manner which is unpredictable for the prover before the attestation. Whenever the attested memory is tampered, the malicious prover thus need to take extra time to recover the original memory. In order to prevent the malicious prover from cheating, every memory address should be accessible in the checksum computation. Additionally, the registers should be fully occupied as well. In this section, we formalize several design principles to ensure fully utilization of the memory and registers in the checksum computation routine.

Choosing Random Function. During the checksum computation, Gen is a random function from l_g bits to l_g bits, and $Addr$ converts the l_g bit generators to the l_a bit addresses. Thus, we can take the concatenation of Gen and $Addr$ as a random function from l_g bits to l_a bits. Since all possible addresses should be accessible when the generators are traversed, proper configuration of the random function in the attestation scheme becomes non-trivial. We discuss two kinds of randomization functions in this work, i.e., the *hash oracle* and the *encryption oracle*.

The *hash oracle* receives a bit-string as input and returns a corresponding random bit-string as output. Since every hash output is computed independently, according to the *coupon collector's problem*, the expected number of independent runs to cover all possible output values grows as $\Theta(t \cdot log(t))$ where t is the number of possible output values. In other words, if the addresses (a_i) and the generators (g_i) have the same length, it is very likely that some memory addresses are uncovered. For instance, when the hash function *SHA-2* is used and both of the generator and the memory address have

the same length of 32bit, only 64% of the addresses can be covered on average when the generators are traversed in our experiments. By enumerating all possible generators in the preparation phase, the adversary may find sufficient *uncovered* addresses and use them to store the malicious code. As a consequence, when hash oracle is used in the attestation protocols, the number of generators should be much larger than the number of addresses. By applying the tail estimate to the *coupon collector's problem*, we can calculate the probability lower-bound of covering all addresses under attestation as $1-(m_d+m_p)^{1-2^{l_g}/((m_d+m_p)\cdot log(m_d+m_p))}$.

On the other hand, the *encryption oracle* can be used to generate random numbers as well by revealing the encryption key to the public. Since the encryption oracle is bijective, all of the memory addresses should be covered in the generator traversal when the generator length is not less than the address length. As a result, the *encryption oracle* becomes very suitable for the random number generation in software-based attestation. Two heavily used implementations of the encryption oracle in the software-based attestation protocols are the stream cipher RC4 and the T-function [13]. RC4 is chosen as the PRNG in SWATT [5] because of its extreme efficiency and compact implementation in the embedded devices. Meanwhile, T-function can produce a single cycle, which ensures the traversal of generators. Thus, it is employed in ICE scheme proposed in ICUBA [7]. A widely used T-function is $x \leftarrow x + (x^2 \vee 5)$ where $\vee$ is the bitwise *or* operator.

Full Address Coverage at Runtime. Even though the addresses can be fully covered in the generator traversal, the actual address coverage is also related to the number of addresses generated at the runtime, which is decided by the number n in the checksum computation routine (Figure 2) and the repeat time k of the consecutive checksum computation (Phase 2). According to the *coupon collector's problem*, in order to fully traverse the whole memory space in the attestation procedure, the minimal number of memory access $n \cdot k$ should satisfy

$$Pr[n \cdot k > c \cdot (m_d + m_p) \cdot log(m_d + m_p)] \leq (m_d + m_p)^{1-c}. \tag{1}$$

Full Register Occupation. According to several existing works [5, 7, 10], the registers in $\mathcal{P}$ are frequently used to store the checksum results. During every iteration in the checksum computation, one of them gets updated to a new value. When any register is unused in the attestation, the malicious prover can exploit it to conduct attacks. Thus, all the registers should be occupied. Moreover, the registers should be chosen in a random order so the malicious prover cannot predict which one is used next. Let the total number of registers used for storing the checksum be u. According to the *coupon collector's problem*, the probability of covering all registers in the checksum computation is lower-bounded by $1 - u^{1-n/(u \cdot log(u))}$.

3.2 $\mathcal{P}_c$ Follow Checksum Computation Routine: Memory Recovering Attack

Given a genuine prover $\mathcal{P}_g$ with the memory state S_g and a compromised prover $\mathcal{P}_c$ with the memory state S_c, the probability of distinguishing their states with a single memory access depends on two factors. The first factor is the percentage of the differences between S_g and S_c, which could be defined as $\lambda(S_g, S_c) = Pr[Read(S_g, a) \neq$

$Read(S_c, a)|a \in \{0,1\}^{l_a}]$. When $\lambda(S_g, S_c)$ is sufficiently large, we can easily detect the modifications in the memory. The second factor is related to the memory content bias in $\mathcal{P}_g$. For instance, the program in $\mathcal{P}_g$ usually contains a large amount of duplicated assembly code such as *mov*, *jmp*, *call*, *cmp*, *nop*, etc. These assembly code can be approximated with high probability. As a consequence, the compromised prover can overwrite the biased memory content into malicious code and recover the original content using a recovering algorithm $\mathcal{C}$ with high probability. Assume the overwriting algorithm is $\mathcal{W}$, the minimal overwriting potion is μ, and memory recovering time $d_{\mathcal{C}}$ is no more than $\delta \cdot d_{Read}$ as required, we could calculate the optimal success probability of the memory recovery as

$$\begin{aligned}\mathbb{P}_m(S, \mu, \delta) = max_{\mathcal{C},\mathcal{W}}\{Pr[Read(S, a) = \mathcal{C}(\mathcal{W}(S), a) \\ | \ a \in \{0,1\}^{l_a}]| \ \delta \cdot d_{Read} \geq d_{\mathcal{C}} \wedge \lambda(S, \mathcal{W}(S)) \geq \mu\}\end{aligned}$$

for any recovering algorithm $\mathcal{C}$ and overwriting algorithm $\mathcal{W}$. δ is the allowed timing overhead for the recovering algorithm comparing with the $Read$ operation. We will discuss more about δ in Section 3.4. When $\delta \geq 1$, we can always implement the recovering algorithm $\mathcal{C}$ for any S as $\mathcal{C}(S, a) = Read(S, a)$, so $\mathbb{P}_m(S, \mu, \delta) \geq 1 - \mu$.

Since $\mathcal{P}_c$ needs to recover the memory content for n times in the checksum computation routine, he can compute the correct checksum if either the memory is recovered successfully for every iteration or the computed checksum collides with the correct one. So overall success probability for $\mathcal{P}_c$ is $\mathbb{P}^n_m(S, \mu, \delta) + (1 - \mathbb{P}^n_m(S, \mu, \delta)) \cdot 2^{-l_r}$. As can be seen from the formula, the success probability is lower-bounded by 2^{-l_r}. So increasing n becomes less significant when n becomes larger. As a consequence, we can define a threshold η for the potential probability increase and then give a lower-bound to the n used in the checksum computation.

$$\mathbb{P}^n_m(S, \mu, \delta) \cdot (1 - 2^{-l_r}) \leq \eta \quad \Longrightarrow \quad n \geq \frac{log(\eta) - log(1 - 2^{-l_r})}{log(\mathbb{P}_m(S, \mu, \delta))} \tag{2}$$

In this work, we suggest to set $\eta = 2^{-l_r}$ which is the success probability's lower-bound. Additionally, we recommend the attestation protocols to set n as the lower-bound given in formula (2) for efficiency and conduct the checksum computation phase (Phase 2) for multiple times to give better security guarantee.

Full Randomization of Data Memory. In the first phase of the generic attestation scheme, $\mathcal{V}$ asks $\mathcal{P}$ to overwrite its data memory with S'_d filled with noise. The unpredictability of S'_d enforces $\mathcal{P}$ to erase its data memory completely. A similar design is taken in [14], but its S'_d is generated by $\mathcal{P}$ using a PRNG seeded by a challenge sent from $\mathcal{V}$. As we discussed above, the recovering algorithm can use the PRNG to generate the memory state with the received challenge at runtime, so $\mathcal{P}_c$ can trade the computation time for memory space. As a result, $\mathcal{P}_c$ can keep the malicious code in its memory, but still produce a valid checksum. In Section 3.4, we show that the checksum computation can have overhead to a degree, so this attack is practical. We thus emphasize that S'_d should be fully randomized by $\mathcal{V}$.

3.3 $\mathcal{P}_c$ Pre-compute Checksum: Challenge Buffering Attack

The attestation scheme is trivially vulnerable to challenge buffering attack that stores the challenge-response pairs directly in the memory. Upon receiving a particular challenge from $\mathcal{V}$, $\mathcal{P}_c$ looks for the corresponding checksum from its memory without computation. Since S'_d and o are received in the attestation procedure, the challenge-response stored in the memory is the tuple $\langle o, r_n \rangle$ which has the length of $l_o + l_r$. Thus, the memory can hold $m \cdot l_c/(l_o + l_r)$ records at most. Additionally, we have 2^{l_o} different receivable values. When $\mathcal{P}_c$ cannot find the record, he can choose a random response from $\{0,1\}^{l_r}$. As a consequence, the probability of computing the correct response with challenge buffering attack method for $\mathcal{P}_c$ can be expressed as follows.

$$\mathbb{P}_b(l_o, l_c, l_r, m_d, m) = b + \frac{1-b}{2^{l_r}} \text{ where } b = \frac{m \cdot l_c}{(l_o + l_r) \cdot 2^{l_o}} \tag{3}$$

As can be seen, $\mathbb{P}_b(l_o, l_c, l_r, m_d, m)$ is also lower-bounded by 2^{-l_r}. So we make the similar suggestion for formula (3) as in Section 3.2 that $b \cdot (1 - 2^{-l_r}) \leq 2^{-l_r}$.

3.4 $\mathcal{P}_c$ Forward Checksum Computation to $\mathcal{A}$: Proxy Attack

As reported in [10], the software-based attestation is particular vulnerable to the proxy attack, in which the compromised prover $\mathcal{P}_c$ forwards the challenge to the adversary $\mathcal{A}$ (a base station) and asks $\mathcal{A}$ to compute the checksum for it. In order to prevent the proxy attack, the expected checksum computation time should be no larger than a time bound, so that $\mathcal{P}_c$ does not have time to wait for the response from $\mathcal{A}$. However, one assumption should be made that $\mathcal{A}$ cannot personate $\mathcal{P}_c$ and communicate with $\mathcal{V}$ directly. Otherwise, the software-based attestation is trivially broken. The assumption can be hold when $\mathcal{V}$ is connected to $\mathcal{P}_c$ using special channels (e.g., bus, usb) that $\mathcal{A}$ has no direct access to.

Assume the network RTT between $\mathcal{V}$ and $\mathcal{P}_g$ varies from $T_{\mathcal{V}}^{min}$ to $T_{\mathcal{V}}^{max}$ and the honest prover $\mathcal{P}_g$ can finish the checksum computation with time $d_g = n \cdot (d_{Gen} + d_{Addr} + d_{Read} + d_{Chk})$, the timing threshold d_{th} on the verifier side thus should be configured as

$$d_{th} \geq d_g + T_{\mathcal{V}}^{max} \tag{4}$$

to ensure the correctness of the attestation protocol defined in Section 2.1. Hence, the maximum usable time for $\mathcal{P}_c$ can be defined as $d_c(T) = d_{th} - T$, where $T \in [T_{\mathcal{V}}^{min}, T_{\mathcal{V}}^{max}]$ is the real network latency between $\mathcal{P}_c$ and $\mathcal{V}$.

On one hand, $\mathcal{P}_c$ could use $d_c(T)$ to conduct the proxy attack. If the network RTT between $\mathcal{A}$ and $\mathcal{P}_c$ varies from $T_{\mathcal{A}}^{min}$ and $T_{\mathcal{A}}^{max}$, in order to prevent the proxy attack completely, we need to make sure that $d_c(T_{\mathcal{V}}^{min}) < T_{\mathcal{A}}^{min}$, which means the proxy attack cannot be conducted even under the optimal RTT for $\mathcal{P}_c$. Thus, the attestation time for the genuine prover should be constrained by

$$d_{th} < T_{\mathcal{A}}^{min} + T_{\mathcal{V}}^{min}. \tag{5}$$

Parameters	SWATT	SCUBA	VIPER
l_o, l_g, l_r (bit)	2048[a], 16, 64	128, 16, 160	-, 32, 832
l_c, l_a (bit)	8, 14	8, 7	8, 13
m_d, m_p, m (unit)	1K, 16K, 17K	0K, 512, 58K	0K, 8K, 4120K
$T_{\mathcal{A}}^{min}, T_{\mathcal{A}}^{max}$	-	$\leq$ 22ms, 51ms	1152ns(43.34ms)[b], 44.10ms
$T_{\mathcal{V}}^{min}, T_{\mathcal{V}}^{max}$	-	$\leq$ 22ms, 51ms	1375ns, 1375ns
d_{th}, d_g	-, 1.8s	2.915s, 2.864s	2300ns, 827ns
n, k, u	3.2E+05, 1, 8	4.0E+04, 1, 10	3, 300, 26

Table 2: Settings of Software-based Attestation Protocols Studied in Section 4

[a] This value is absent in [5] and assigned by us. The justification is made in Section 4.1.

[b] The RTT in the parentheses is the real network latency collected in the experiments of [10]. The RTT in front is the theoretical lower-bound used in [10].

On the other hand, $\mathcal{P}_c$ could use $d_c(T)$ to conduct the memory recovering attack. So we calculate the δ specified in the *memory recovery attack* as follows.

$$\frac{d_{Gen} + d_{Addr} + \delta \cdot d_{Read} + d_{Chk}}{d_{Gen} + d_{Addr} + d_{Read} + d_{Chk}} = \frac{d_c(T)}{d_g} = \frac{d_{th} - T}{d_g} \tag{6}$$

Since, $\delta \propto d_g^{-1} \propto n^{-1}$, in order to keep the δ small, the checksum computation routine should use the largest n as possible, when formula (4) and (5) are still satisfied.

4 Case Studies

In this section, we analyze three well-known software-based attestation protocols, i.e., SWATT [5], SCUBA [7] and VIPER [10]. Since the generic software-based attestation scheme is configured with the parameters listed in Table 1, we first extract them from the real protocols as shown in Table 2. As can be seen, our generic attestation scheme can capture existing software-based attestation protocols readily. Then, we apply the security criteria described in Section 3 manually to the extracted parameters to find security flaws. In the following subsections, we briefly introduce the protocols first, and then give detailed vulnerabilities and justifications grouped by the topics in **bold** font. We mark the topics with " $\star$ " if they are reported for the first time in the literature.

4.1 SWATT

SWATT [5] randomly traverses the memory to compute the checksum. Its security is guaranteed by the side channel on time consumed in the checksum computation. SWATT does not consider network RTT, so we do not discuss time related properties for SWATT. In addition, SWATT uses RC4 as the PRNG and takes the challenge as the seed of the RC4. As the length of the challenge chosen in the SWATT is not mentioned in [5], we assume that the challenge is long enough to fully randomize the initial state of RC4, which means $l_o = 256 \cdot 8$ bits.

Unattested Data Memory. The micro-controller in SWATT has $16KB$ program memory and $1KB$ data memory. Based on the analysis of the generic attestation scheme, SWATT is insecure because it neither has Secure Erasure Phase to overwrite the data memory nor uses any additional complement to secure the data memory. In fact, the authors of SWATT assumed in [5] that non-executable data memory can do no harm to the security of software-based attestation by mistake. In [9], Castelluccia et al. point out that the data memory should be verified in SWATT, otherwise the protocol is vulnerable to the ROP [15, 16] attack. In this work, we suggest to securely erase the data memory in SWATT by following our generic attestation scheme.

***Too Large Iteration Number for Computing One Checksum.** The main loop of SWATT has only 16 assembly instructions, which takes 23 machine cycles. Inserting one if statement in the loop will cause additional 13% overhead. As a result, we assume that the recovering algorithm $\mathcal{C}$ only has time to read the memory content as $Read$ does without doing any extra computation. Hence, the success probability of the memory recovering of SWATT becomes $\mathbb{P}_m(S, \mu, \delta) = 1 - \mu$, where μ is the percentage of the modified memory. According to the formula (2), after setting η as suggested, we have $n \geq -64/log(1-\mu)$. When $\mu = 0.001$ which left only 16 byte memory for the adversary, we should set n as 44340, which is much smaller than the iteration number 320000 used in SWATT. In order to increase the difficulty of attacking the attestation protocol and traverse the memory address in the platform, more rounds of checksum computation could be conducted. According to formula (1), when $\mu = 0.001$, $n = 44340$ and $c = 2$ (the same setting in SWATT), we have $k \geq 11$. So we should conduct the checksum computation for 11 times. By using this new configuration, the overall memory access time is approximately the same as SWATT while security guarantee becomes dramatically better.

4.2 SCUBA

SCUBA [7] is a software-based attestation protocol that based on Indisputable Code Execution (ICE). Rather than attesting the whole memory, the ICE offers security guarantee by only verifying a small portion of the code. The $Read$ and Chk implemented in the ICE scheme are different from those given in Section 2.2. However, they can be generalized into our framework. In SCUBA, $Read$ not only reads the memory content, but also returns the Program Pointer (PC), the current address, the current generator, the loop counter and other registers. The Chk function then computes the checksum based on all of them. In order to compute the correct checksum for the modified attestation routine, the malicious prover has to simulate the execution for all of them, which thus lead to large and detectable overhead on the computation time. If the malicious prover do not change the attested code, the attested code can update the prover's whole memory to a genuine state so the malicious code shall be removed from the prover.

***Proxy Attack is Indefensible.** In SCUBA, network RTT is explicitly evaluated in the experiment as summarized in Table 2. The prover in SCUBA communicates with the verifier over wireless network. Even though the adversary is assumed to be physically absent during the attestation in SCUBA, this assumption seems to be too strong to be

hold when a wireless network presents. Thus, we give a detailed analysis for the proxy attack to SCUBA as follows.

According to [7], the maximum network RTT is $51ms$ in SCUBA. By observing the experiment results, the minimum network RTT should be no larger than $22ms$. As the adversary and the verifier share the same wireless network, the network latency for their communication with the prover should be indifferent. So we have $T_{\mathcal{A}}^{min} = T_{\mathcal{V}}^{min} \leq 22ms$ and $T_{\mathcal{A}}^{max} = T_{\mathcal{V}}^{max} = 51ms$. According to formula (4), we have $d_{th} \geq d_g + T_{\mathcal{V}}^{max} \geq 51ms$. On the other hand, according to formula (5), we have $d_{th} < T_{\mathcal{A}}^{min} + T_{\mathcal{V}}^{min} \leq 44ms$. Hence, we cannot find a valid threshold d_{th} from this network configuration. When the adversary presents in the attestation, the proxy attack thus cannot be defended by SCUBA without additional assumptions.

Moreover, if the verifier does not communicate with the prover with a secure channel (e.g., the verifier uses the wireless network to the communicate with the prover in this case), the adversary can personate the prover and send the checksum to the verifier directly. Since the adversary can compromise the prover, he can obtain the secret key stored in the prover as well. So encrypting the wireless channel will not work. We suggest that the verifier should communicate with the prover in an exclusive method, such as the usb connection, which is also inexpensive. More importantly, the adversary cannot use this communication method as it is highly controllable.

Security Claim Justification. Our framework can not only be used to find potential vulnerabilities, but also give justifications to the security claims made in existing works. In SCUBA [7], the malicious prover may exploit the network latency to conduct memory recovering attack without being detected. However, if the timing overhead of the attack is even larger than the largest network latency, the attack then becomes detectable. According to this, the authors of SCUBA claim that the checksum computation time adopted in SCUBA can always detect the memory copy attack, which is the most efficient memory recovering attack method known to the authors, even if the malicious prover can communicate without network delay.

In this work, we can justify their security claim with our framework. When the proxy attack is not considered in SCUBA, increasing the checksum computation time does not introduce vulnerability. According to formula (6), we have $d_c(T)/d_g = (d_{th} - T)/d_g$. The experiment results in [7] show that the memory copy attack is most efficient attack which introduces 3% overhead to the checksum computation. In order to detect the memory copy attack, we should ensure that $\forall T \in [T_{\mathcal{V}}^{min}, T_{\mathcal{V}}^{max}], d_c(T)/d_g < 1.03$. As we assume that the malicious prover can communicate without network delay, we set $T_{\mathcal{V}}^{min}$ as 0. By applying formula (4), we have $d_g > 1700ms$. Since d_g chosen in SCUBA is indeed larger than $1700ms$, the security claim made by the authors is valid.

4.3 VIPER

VIPER [10] is a software-based attestation scheme designed to verify the integrity of peripherals' firmware in a typical x86 computer system. They are proposed to defend all known software-based attacks, including the proxy attack.

***Absence of Random Function.** VIPER uses a similar design as ICE scheme, while its generators are not produced by a PRNG during the checksum computation, which

does not comply to our generic attestation scheme. The authors implement the checksum function into 32 code blocks. One register is updated in every code block with the memory content and the program counter (PC). Both of the code block and the memory address are chosen based on the current checksum. Thus, the randomness of the checksum is purely introduced by the PC and the memory content. However, the PC is incremented in a deterministic way inside each code block and the memory content usually is biased as illustrated in Section 3.2. As the randomness could be biased, the adversary can traverse all challenge values and he may find some memory addresses that are unreachable for the checksum computation routine, as we discussed in Section 3.1. Hence, the security provided by VIPER is unclear.

⋆**Insufficient Iteration Number.** In VIPER, the number of iterations used in the checksum computation routine is only 3, which leads to at least 23 unused registers in the attestation. Vulnerabilities may be introduced as discussed in Section 3.1. Even if the registers are chosen in a fully randomized manner and the adversary cannot predict which register will be used beforehand, the malicious prover still has a high probability to use some registers without being detected. In fact, two or even one register could be enough for conducting an attack in practice.

5 Related Works

A large amount of software-based attestation protocols have been designed and implemented [17, 5, 18, 6, 12, 7, 19–22, 10, 23]. Specifically, SWATT [5] is a software-based attestation scheme that uses the response timing of the memory checksum computation to identify the compromised embedded devices. In order to prevent replay attack, the prover's memory is traversed in SWATT in a random manner based on a challenge sent from the verifier. Rather than attesting the whole memory content, SCUBA [7] only checks the protocol implemented in the embedded devices and securely updates the memory content of the embedded devices after the attestation is finished successfully. It is based on the ICE (Indisputable Code Execution) checksum computation scheme, which enables the verifier to obtain an indisputable guarantee that the SCUBA protocol will be executed as untampered in the embedded devices. VIPER [10] is later proposed to defense against the adversary who can communicate with the embedded devices during the attestation. Network latency is consider in VIPER to prevent the proxy attack. Perito et al. [22] develop a software-based secure code update protocol. It first overwrites the target device's whole memory with random noise and then asks the target device to generate a checksum based on its memory state. The target device could generate the correct checksum only if it has erased all its memory content, so the malicious code should also be removed. Besides the attestation protocol designed for resource constrained devices, Seshadri et al. [12] develop the software-based attestation protocol named Pioneer for the Intel Pentium IV Xeon Processor with x86 architecture.

However, the software-based attestation protocol design is challenging and error-prone [8, 9]. Hence, it becomes necessary and urgent to develop an analysis framework for the attestation protocol design. Armknecht et al. [24] recently provide a security framework for the analysis and design of software attestation. In their work, they assume the cryptographic primitives such as Pseudo-Random Number Generators

(PRNGs) and hash functions might be insecure and give a upper-bound to the advantage of the malicious prover in the attestation scheme. They mainly consider six factors: (1) the memory content could be biased; (2) the memory addresses traversed in the checksum computation may not be fully randomized; (3) the memory addresses could be computed without using the default method; (4) the correct checksum could be computed without finishing the checksum computation routine; (5) the checksum could be generated without using the default checksum computation function; (6) the challenge-response pairs could be pre-computed and stored in the memory. In this work, we do not consider factor (2-5) based on two reasons. First, the attestation routine used in the protocol can be updated at runtime, so we can always update the cryptographic functions to meet the higher security standard and requirement. For instance, since the hash function like *MD5* is insecure nowadays, we can replace it with *SHA-2* or *SHA-3* to reclaim security. More importantly, the upper-bounds of the factor (2-5) are very hard to measure in practice. For example, given a well-known weak hash function like *MD5*, it is hard to measure the *time-bounded pseudo-randomness*, corresponding to factor (2), defined in [24]. Comparing with [24], we additionally consider observable network latency, stronger threat model, unpredictable data memory, several security criteria and various attack schemes. More importantly, our framework has been successfully applied to several existing software-based attestation protocols to find vulnerabilities.

6 Discussions and Future Works

In this work, we present a practical analysis framework for software-based attestation scheme. We explicitly consider the network latency and the data memory in the system. Furthermore, the adversary presented in this work can not only reprogram the compromised provers before the attestation but also communicate with them during the attestation. We successfully apply our framework to three well-known software-based attestation protocols manually. The results show that our framework can practically find security flaws in their protocol design and give justifications to their security claims.

The deployment environment, including device architecture, network environment, efficiency requirement, etc. usually complicates the correctness of the software-based attestation protocols. Specifically, identifying the most effective overwriting and recovering algorithms becomes very hard, which limits the application of our framework. For future works, we believe that fine-grain measurement for the overwriting and recovering algorithms in the practical application context is useful. Another future work is investigating the impact of timing requirement when the attestation efficiency is concerned. In this work, we assume that software-based attestation can take as much time as it needs. Nevertheless, in reality, we may require the attestation protocols to be finished within a timing threshold. Hence, the probability of identifying the compromised prover will be affected, and choosing the right configurations (e.g., the iteration number) then becomes more challenging.

References

1. W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *S&P*. IEEE CS, 1997, pp. 65–71.

2. P. England, B. W. Lampson, J. Manferdelli, M. Peinado, and B. Willman, "A trusted open platform," *IEEE Computer*, vol. 36, no. 7, pp. 55–62, 2003.
3. R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a tcg-based integrity measurement architecture," in *USENIX Security*. USENIX, 2004, pp. 223–238.
4. C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence," in *DSN*. IEEE, 2009, pp. 115–124.
5. A. Seshadri, A. Perrig, L. van Doorn, and P. K. Khosla, "Swatt: Software-based attestation for embedded devices," in *S&P*. IEEE CS, 2004, pp. 272–282.
6. M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim, "Remote software-based attestation for wireless sensors," in *ESAS*. Springer, 2005, pp. 27–41.
7. A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. K. Khosla, "Scuba: Secure code update by attestation in sensor networks," in *WiSe*. ACM, 2006, pp. 85–94.
8. U. Shankar, M. Chew, and J. D. Tygar, "Side effects are not sufficient to authenticate software," in *USENIX Security*. USENIX, 2004, pp. 89–102.
9. C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices," in *CCS*. ACM, 2009, pp. 400–409.
10. Y. Li, J. M. McCune, and A. Perrig, "Viper: verifying the integrity of peripherals' firmware," in *CCS*. ACM, 2011, pp. 3–16.
11. "Trusted Platform Module." [Online]. Available: http://www.trustedcomputinggroup.org/developers/trusted_platform_module
12. A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. K. Khosla, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems," in *SOSP*. ACM, 2005, pp. 1–16.
13. A. Klimov and A. Shamir, "New cryptographic primitives based on multiword t-functions," in *FSE*, ser. LNCS, vol. 3017. Springer, 2004, pp. 1–15.
14. Y.-G. Choi, J. Kang, and D. Nyang, "Proactive code verification protocol in wireless sensor network," in *ICCSA*, ser. LNCS, vol. 4706. Springer, 2007, pp. 1085–1096.
15. H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *CCS*. ACM, 2007, pp. 552–561.
16. E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: generalizing return-oriented programming to risc," in *CCS*. ACM, 2008, pp. 27–38.
17. R. Kennell and L. H. Jamieson, "Establishing the genuinity of remote computer systems," in *USENIX Security*. USENIX, 2003, pp. 21–21.
18. J. T. Giffin, M. Christodorescu, and L. Kruger, "Strengthening software self-checksumming via self-modifying code," in *ACSAC*. IEEE CS, 2005, pp. 23–32.
19. Y. Yang, X. Wang, S. Zhu, and G. Cao, "Distributed software-based attestation for node compromise detection in sensor networks," in *SRDS*. IEEE CS, 2007, pp. 219–230.
20. R. W. Gardner, S. Garera, and A. D. Rubin, "Detecting code alteration by creating a temporary memory bottleneck," *IEEE Trans. Inf. Forensics Security*, vol. 4, no. 4, 2009.
21. T. AbuHmed, N. Nyamaa, and D. Nyang, "Software-based remote code attestation in wireless sensor network," in *GLOBECOM*. IEEE, 2009, pp. 1–8.
22. D. Perito and G. Tsudik, "Secure code update for embedded devices via proofs of secure erasure," in *ESORICS*, ser. LNCS, vol. 6345. Springer, 2010, pp. 643–662.
23. X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth, "New results for timing-based attestation," in *S&P*. IEEE CS, 2012, pp. 239–253.
24. F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, "A security framework for the analysis and design of software attestation," in *CCS*. ACM, 2013, pp. 1–12.