Code Attestation with Compressed Instruction Code

Benjamin Vetter and Dirk Westhoff

Department of Computer Science, Hamburg University of Applied Sciences (HAW)
Berliner Tor 7, 20099, Hamburg
{benjamin.vetter, dirk.westhoff}@haw-hamburg.de

Abstract: Available purely software based code attestation protocols have recently been shown to be cheatable. In this work we propose to upload *compressed* instruction code to make the code attestation protocol robust against a so called compresssion attack. The described secure code attestation protocol makes use of proposed microcontroller architectures for reading out compressed instruction code.

1 Introduction[1]

The evolution of the ubiquitous computing vision towards full-fledged real world applications faces a diversity of new problems. Due to the fact that for many applications cost-efficient hardware is an issue, one can not guarantee that a code image that has been uploaded on a non-tamper resistant device will always run in a correct and un-manipulated way. Even worse, it may behave in a Byzantine manner such that the device sometimes behaves correctly and sometimes behaves incorrectly. One strategy to detect such misbehaving nodes in a sensor network is to run from time to time a challenge-response protocol between the restricted device and a master device - the verifier - that is sending the challenge. However, recently it has been shown that purely software based code attestation [SLP+06], [SPD+04], [SMK+05] is vulnerable against a set of attacks. The rest of the paper is organized as follows: Section 2 introduces related work. Section 3 presents the adversary model and describes the so called compression attack the attacker can perform to break recently proposed code attestation protocols. In Section 4 we propose our countermeasure to deal with compression attacks and in Section 5 we give insights on how to execute compressed instruction code as necessary requirement for this approach. Section 6 discusses suitable compression algorithms and in Section 7 we provide the security analysis of the proposed solution. Conclusions and open issues are presented in Section 8.

[1] An extended version of this work has been uploaded to the Computing Research Repository (CoRR).

2 Related Work

One can subdivide code attestation techniques into two subsets: the first class of approaches is using challenge-response protocols in conjunction with harsh timing restrictions for the restricted device's response. E.g., SWATT [SPD+04], proposed by Seshadri et al., belongs to this group. In SWATT, the verifier measures the time a node needs to calculate a checksum of its memory contents, i.e. how long the node needs to respond to a challenge. As compromised nodes are supposed to require more time to reply, because they have to hide their code injections from the checksum, compromised nodes are supposed to be detectable. The second proposed class of countermeasures tries to prevent attackers from getting space to inject their bogus code in an undetected manner. E.g., Yang et al. propose to fill empty program memory with pseudorandomly generated noise that is also fed into the response [YWZ+07]. Therefore, an attacker can not inject his bogus code into the noise in an undetected manner. However, as we will see, Castelluccia et al. [CFP+09] have shown that both types of aformentioned countermeasures can be circumvented. Later we provide more details on this.

3 Adversary Model and Compression Attack

After node deployment and before the first round of the attestation protocol starts, the attacker has full control over all device memories such that he can modify program memory or any other memories like e.g. the external memory. At attestation time, when the challenge-response based attestation protocol is running, the attacker has no physical control over the restricted device anymore. However, the device may yet run malicious code. It is up to the code attestation protocol to detect this independently of the fact that the attacker may find ways to store the original uploaded code image at a different memory than the program memory. Note that we do not consider fluctual data memory. *Control Flow Integrity* could prevent attacks that use techniques like *Return-Oriented Programming* [CFP+09], [ABE+05], [FGS09]. Obviously, during the phase in which the attacker has full control over the restricted device, the attacker is also able to either modify the code for the code-attestation protocol itself or to read out any sensitive data, like pre-shared keys, in case the protocol is based on this.

One major challenge for a purely software-based code attestation for embedded devices is the so called compression attack. This attack cheats a basic challenge-response based code attestation as follows: the originally uploaded program, which shall temporarily be checked by the attestation protocol to be exclusively stored in the program memory, is subsequently compressed by the attacker. Depending on the concrete compression algorithm and the actual uploaded code image, the compression gain ranges from 12% up to 47% [CFP+09]. An attacker can use such free program memory to store and run bogus code on the node's program memory. Current solutions for secure code-attestation prevent an attacker from using the free program memory by filling it with pseudorandomly generated words instead of the default entry 0FF [YWZ+07]. Since the aforementioned pseudorandomly generated words are required to be part of the response of a code at-

testation protocol, the verifier needs to know respectively may be able to compute such pseudorandomly generated words.

However, Castelluccia et al. have shown that cheating such kinds of attestation protocols is still possible: whenever the restricted device (prover) receives a nonce from the master device (verifier), it decompresses the earlier compressed original program on-the-fly and subsequently computes the hash value $x = h(nonce||CI||PRW)$ by applying the hash function $h()$. The x is the response of the challenge-response protocol. The CI denotes the originally uploaded code image, and the PRW is the pseudo-randomly filled content within the remaining free program memory at load-time. Obviously this simple challenge-response based code attestation fails: whenever the prover receives a fresh nonce, the attacker *decompresses* the compressed CI. This provides all the relevant input parameters for the computation of the hash function, namely the CI, the nonce and the PRW such that the master device subsequently receives the response x within a given time interval which it verifies to be correct. Finally note that, to save his own bogusly uploaded code image $\widetilde{CI}$, the attacker could have stored $\widetilde{CI}$ also within the external memory. Subsequently to the time-critical code attestation phase, he has enough time to again compress the CI and read $\widetilde{CI}$ from external memory to program memory.

4 Attestation of Compressed Instruction Code

Our countermeasure against uploading malicious code into the program memory and subsequently not being able to detect this, uses i) a hardware extension at the micro-controller, and ii) a strict policy for uploading CIs into the program memory. This policy is to only upload a yet *compressed* code image $C(CI)$ into the program memory and to fill the remaining part with PRW[2]. Consequently, the attacker cannot allocate such easily free program memory anymore to tracelessly upload malicious code by applying the above described compression attack. Note that the challenge (a fresh nonce), which goes into the hash computation for every run of the code-attestation anew, enforces the prover to always compute the hash value (response) with a compressed CI and PRW anew. In our proposed setting the response x thus is computed as $h(nonce||C(CI)||PRW)$ where the C is a properly chosen lossless data compression algorithm. More details on the C and other refinements on $C(CI)$ will be provided later. The adapted code attestation protocol is shown in Figure 1 (Option 1).

Please note that our protocol's intention is to raise the attacker's overhead for passing the attestation by orders of magnitude. Therefore, it is still essential for our protocol to enforce a runtime restriction to detect the attacker. We term ϵ as the duration of the time interval $[t_0, t_1]$ measured by the local clock of the verifier. The t_0 denotes the sending time of the challenge $nonce$ and the t_1 denotes the receiving time of the response x. We emphasize that a proper choice of the threshold T_{em} with $\epsilon < T_{em}$ is prover device-dependent.

[2]The PRW can not be compressed [YWZ+07]. In fact $C(PRW)$ would result in $|C(PRW)| \geq |PRW|$ eventually providing another attack vector to save memory by computing $C^{-1}(C(PRW))$.

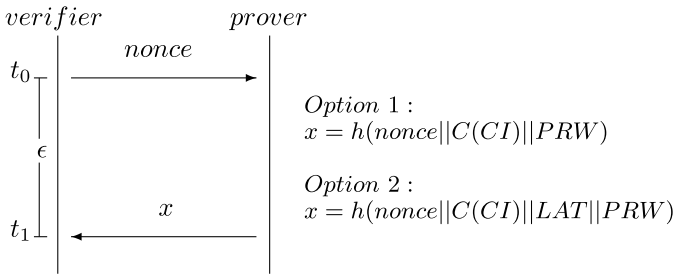

Figure 1: Derivates of the secure code attestation protocol with lossless data compression algorithm.

5 Execution of Compressed Instruction Code

One remaining problem with this approach is how to run compressed code? To solve this issue one needs to incorporate a hardware extension at the micro-controller. Please note that the approach to upload a compressed code image into the program memory is not new. It has recently been proposed by Yamada et al. [YFN+06]. Early work on this can be found in [WC92]. However, originally it has been proposed with the objective to offer a high compression ratio and a fast instruction expendability - and not as a building block to protect against a bogus code image in the program memory, like we are proposing.

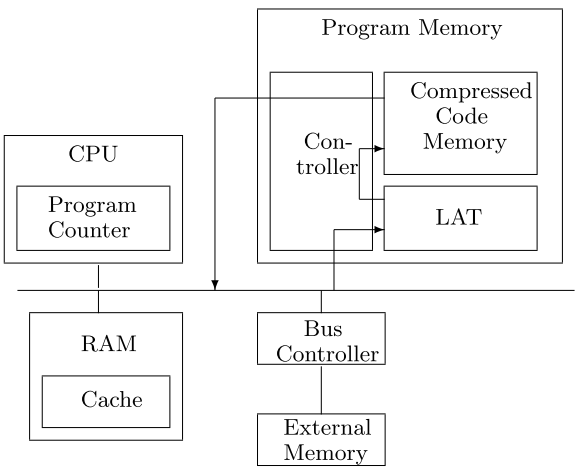

Figure 2: Micro-controller architecture with a compressed code memory, a LAT and a cache[3].

Figure 2 illustrates the architecture that can be used to decompress compressed code blocks on-the-fly, as proposed by Wolfe et al. [WC92]. Xu et al. proposed to store the cache within the RAM [XJ03]. The code image CI is partitioned into blocks of equal length,

[3]Please note that a similar design was first proposed by Yamada et al. [YFN+06] with a dictionary memory instead of a LAT and without a cache.

each individually compressed. The compressed code blocks together build the compressed code image $C(CI)$, which is stored within the compressed code memory. As the code blocks are compressed, they are no longer of equal length. Therefore, a data structure, the Line Address Table (LAT), is used to store the block's offsets. The block size s has to be equal to the available cache size ($s = |cache|$). The cache is used to temporarily store the code block that is currently executed. A cache flush occurs when code of the CI has to be executed that is currently not present within the cache. E.g., a jmp instruction to an address outside of the cache's current scope causes a cache-flush. The micro-controller then uses the LAT to decompress the compressed code block that is part of the $C(CI)$ and specified by the jmp instruction, and subsequently stores the decompressed code block in the cache.

So we propose to only allow to load yet compressed code into the program memory and to decompress the code at runtime. This architecture can be used to defend against attacks where free program memory space can be generated by compressing the originally uploaded code image and filling this gap with malicious code (including the compression/decompression function). A code attestation protocol based on simply hashing the original code image plus the remaining free program memory space would not detect this.

Some Remarks: To be able to subsequently decompress the CI at runtime we are not allowed to compress the LAT itself. However, in section 7 we show that the attacker does not succeed in sufficiently compressing the LAT. The LAT as well as the compressed code memory are *regions* within the program memory. Consequently, an attacker could either fully overwrite or partially modify the LAT. To detect modifications of the LAT we refine the computation of the response x such that (Option 2 in Figure 1):

$$x = h(nonce||C(CI)||LAT||PRW) \tag{1}$$

6 Choice of the Data Compression Algorithm

6.1 Envisioned Properties

The proper choice of a suitable lossless data compression algorithm C is essential with respect to the proposed security architecture. We need to find a lossless data compression algorithm which shall provide the following properties: i) a high compression ratio for a typical CI, and ii) fast decompression.

With respect to property number one we state that it is one of the properties of any lossless data compression algorithm that for *typical* input files, which contain frequently used data chunks, the compression rate is rather high. However, vice versa, if the input file contains many seldomly used data chunks, the resulting compression ratio is rather poor. Moreover, the compression algorithm C_h chosen by the honest party should ideally provide the highest compression rate compared to other compression candidates, e.g. C_a chosen by the attacker. However, our proposed code attestation protocol does not rely on the availability of the best possible compression algorithm, as we will show.

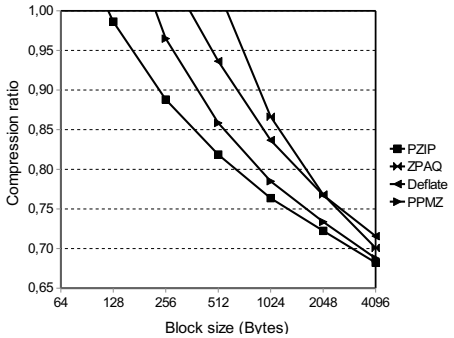

Figure 3: Compression ratios of typical compression algorithms for the multi-hop oscilloscope program image.

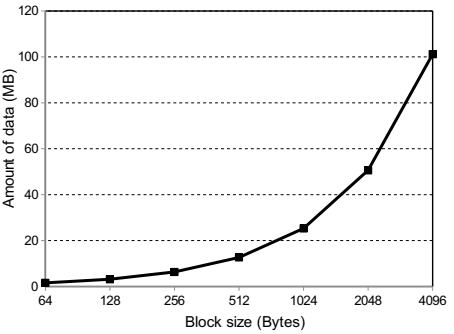

Figure 4: Amount of decompressed data for varying block sizes during the attestation.

The second property is required since decompression of a code image instruction should ideally not delay the execution of the originally loaded program. On the contrary, there is no technical requirement that restricts the compression time before uploading the CI.

6.2 Candidates

Initially we considered *Canonical Huffman Encoding* (CHE) [Hu52] as lossless data compression algorithm C. However, the disadvantage of the CHE for our purposes is its relatively small gain of compression results on MicaZ with on average 12.19% for various typical WSN programs [CFP+09]. For comparison, the lossless data compression algorithm *Prediction by Partial Matching* (PPM) provides an average gain of 47.45% for typical WSN applications. Unfortunately, such a significant gain difference of the compression algorithms CHE and PPM again opens the door for an attack to make use of this gain difference of approximately 35%. The attacker can apply PPM on the compressed code image $C_{CHE}(CI)$ and again generate free space for his own bogus malicious code in either of the two ways: i) $C_a(C_h(CI)) := C_{PPM}(C_{CHE}(CI))$, respectively ii) $C_a(C_h^{-1}(C_h(CI))) := C_{PPM}(C_{CHE}^{-1}(C_{CHE}(CI)))$. C^{-1} denotes the decompression operation. Due to the aforementioned reason we also analyzed Deflate, ZPAQ and further derivates of PPM, namely PZIP and PPMZ.

Figure 3 shows that the chosen algorithms provide varying compression ratios depending on the block size s_h. This is illustrated for our benchmark code image *multi-hop oscilloscope* ($|CI| = 25.9KB$) which ships with TinyOS. Large block sizes provide better compression ratios than small block sizes. If we choose and apply a tuple (C_h, s_h) the attacker can only gain additional free memory $|C_a(C_h(CI))| - |C_h(CI)| = |\widetilde{CI}|$ using block based compression schemes by choosing $s_a > s_h$ if $C_a = C_h$, or otherwise: $s_a \leq s_h$ (for some (C_a, s_a)). Nevertheless, if the attacker chooses a much smaller block size the compression ratio will suffer. Therefore, when we compress the CI with a larger

block size the attacker is forced to use a larger block size as well. Since the decompression of larger blocks increases the overhead, the time necessary for decompression is increased as well, especially on low-performance platforms like sensor nodes. This fact becomes significant if we take into account that the memory traversal during the attestation runs in a pseudorandomly manner with $nonce$ as the seed for a PRNG, which forces a strict ordering of the CI's words when calculating the response x [ANN09]. It forces the attacker to decompress each block approximately s_a times, where s_a denotes the block size chosen by the attacker. Moreover, this disables the attacker to apply a compression algorithm C_a that sacrifices performance for higher compression ratios since the overhead increases for larger block sizes s_a recognizably. Therefore, the use of such algorithms is easily detectable with the choice of a large block size s_h and a threshold T_{pm} as the upper duration for performing compression attacks on the program memory. Obviously, $\epsilon < min\{T_{em}, T_{pm}\}$ with $T_{pm} > T_{em}$ as we will see.

Figure 4 shows the amount of temporarily decompressed data during the attestation, which increases for larger block sizes. The attacker has to read about $s_a \cdot |C_a(CI)|$ bytes from the program memory during the attestation if he compressed the full CI previously. If the attacker chooses the block size to be $s_a = 2048$ bytes and C_a to be PZIP, he will have to read up to $37MB$ from program memory to decompress all blocks s_a times and subsequently be able to calculate x. This is a huge overhead compared to $|CI| = 25.9KB$. Obviously, this huge amount of data is an immense burden for an attacker, in particular on platforms with low bandwidth for reading from program memory. While platforms capable of reading $50MB/s$ result in less than 1 second timing overhead for 2048 byte blocks, platforms capable of reading only $1MB/s$ require up to 40 seconds and thus are easily detectable by the proposed attestation protocol.

Obviously, the overhead to decompress every block s_a times impacts the time necessary for the attacker to calculate the valid response x significantly on restricted platforms. As an uncompromised node doesn't have to calculate $C_h^{-1}(C_h(CI))$ at attestation time, i.e. decompress the compressed program image, the block size enables us to raise and adjust the overhead for the attacker by orders of magnitude. Therefore, we can discover the existence of the attacker reliably through a proper choice for the device-dependent value of ϵ. However, a larger cache size respectively s_h slows down the on-the-fly decompression routine during normal operation of the restricted device. On the other hand, a larger cache decreases the number of cache misses. Therefore, a necessary decompression is more seldom for a larger cache size, but takes more time to complete.

7 Security Analysis

Our security analysis considers four attack vectors, namely decompressing the code image, attacks on the LAT, attacks by using the external memory and DoS attacks.

Decompressing the Code Image. The attacker is able to decrease the timing overhead by exploiting the fact that different blocks can be compressed with different compression ratios. Therefore, the attacker could pick only those blocks which provide the best compres-

sion ratios out of all blocks until he gains sufficient memory to store his bogus code. Since the blocks are yet compressed with a properly chosen lossless compression algorithm, each of them provides a similar compression ratio. To overcome this issue, the attacker could first decompress the compressed CI and compress it for his own afterwards. I.e., the attacker could calculate $C_h^{-1}(C_h(CI))$ and $C_a(C_h^{-1}(C_h(CI)))$ afterwards. During the attestation he then has to calculate $C_h(C_a^{-1}(C_a(CI)))$ to pass the attestation. Therefore, this method further increases the overhead for the attacker, especially if we choose a (C_h, s_h) that compresses rather slowly. Moreover, the attacker's possible gain is expected to be low, because blocks which provide a good compression ratio to the attacker will provide a good compression ratio to us as well.

However, even without calculating $C_h^{-1}(C_h(CI))$ the attacker still requires to compress only as much blocks as he needs to gain enough free memory for the $\widetilde{CI}$. The exact number of blocks an attacker has to use depends on our choice of (C_h, s_h) as well as the attacker's choice of (C_a, s_a) and $|\widetilde{CI}|$ itself. Please note that besides the $\widetilde{CI}$ the attacker has to also store the code of the decompression routine C_a^{-1} and the LAT_a within the program memory. As Castelluccia et al. have to spend 1707 bytes for a huffman decompression routine [CFP+09] used in their compression attack, which is a relatively simple algorithm compared to the compression algorithms proposed in this paper, we force the attacker to compress at least multiple blocks to get a chance to gain enough space for his needs. In general, the attacker has to compress

$$\#Blocks = \frac{|\widetilde{CI}| + |C_a^{-1}| + |LAT_a|}{GainPerBlock} \tag{2}$$

where

$$GainPerBlock = \frac{TotalGain}{\#Blocks_{total}} \tag{3}$$

on average with

$$TotalGain = |C_h(CI)| - |C_a(CI)| \tag{4}$$

and

$$\#Blocks_{total} = \frac{|CI|}{s_a}. \tag{5}$$

The memory overhead then is about $\#Blocks \cdot s_a \cdot \frac{|C_a(CI)|}{|CI|} \cdot s_a$. We assume the attacker has to store at least $1KB$ of data[4], i.e. $|\widetilde{CI}| + |C_a^{-1}| + |LAT_a| = 1KB$ and he will calculate $C_h^{-1}(C_h(CI))$ before compressing the CI for his own. For example, if we choose $(C_h = PZIP, s_h = 512$ bytes) and the attacker chooses $(C_a = PPMZ, s_a = 2048$ bytes) the attacker's memory overhead is about $17.3MB$.

Figure 5 shows the attacker's possible choices for (C_a, s_a) to gain sufficient memory whereas $C_h = PZIP$ with varying s_h is our choice of a compression algorithm. For the attacker's choices we focus on compression algorithms mentioned in this paper only, namely PZIP, PPMZ, ZPAQ and Deflate for block sizes ranging from 64 bytes to 2048 bytes. On platforms capable of reading $1MB/s$ of data from program memory, we argue

[4]Please note that this is a very optimistic value from the attacker's point of view.

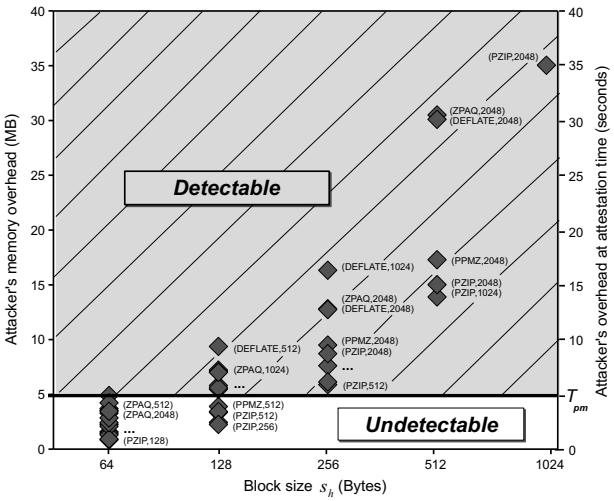

Figure 5: The attacker's possible compression choices for $C_h = PZIP$, a varying s_h and a platform capable of reading $1MB/s$ from program memory.

that memory overhead above $5MB$ is easily detectable since it slows down the attestation for about 5 seconds. Therefore, even if we choose rather small block sizes of $s_h \geq 256$ bytes the attack is still detectable. Please note that we do not even take the CPU overhead into account here. From a security point of view we argue to always use the largest possible block size s_h. In practice cache sizes above $1KB$ are hardly feasible, especially on embedded devices with less than $4KB$ of data memory. Therefore, we propose to choose $(C_h, s_h \geq 512$ bytes$)$. Please note that other combinations will be feasible as well, but one has to choose s_h for other compression algorithms more carefully.

Attacks on the LAT. The countermeasure to the compression attack is the compression of the CI with a suitable data compression algorithm. Thus, the only remaining non-compressed data besides the PRW which has been argued to be not effectively compressable is the LAT_h. Consequently, if the (C_h, s_h) for compressing the CI has been chosen properly, the only remaining compression attack is to compress the LAT_h itself to save program memory $(C_a(LAT_h))$. If the attacker succeeds in saving enough program memory out of this to additionally store a bogus code image $\widetilde{CI}$, and at the same time requires $\epsilon < min\{T_{em}, T_{pm}\}$, the attack is successful and not detectable by our code attestation protocol. However, recall that a lossless data compression algorithm does not provide the same compression ratio for every ingoing uncompressed data; in particular a LAT due to its condensed form can not significantly be compressed as we will see. Moreover, we state that typically it holds $|LAT_h| << |CI|$ and $|CI| \leq |PRW|$[5]. In general, the number of

[5]Typical CI sizes for WSN applications are between 10 to 60KBytes such that the $|PRW|$ occupies between 63 Kbytes to 113 Kbytes [CFP+09].

entries of a LAT can be computed as

$$\#Entries(LAT) = \frac{|CI|}{s} \tag{6}$$

So, even if $C_a(LAT_h)$ and $C_a(CI)$ with (C_a, s_a) would provide the same compression ratio, which obviously is not the case, the absolute gain of program memory for an attacker who purely can compress the remaining uncompressed LAT_h would be significantly smaller. E.g., we assume an embedded device with $128KB$ of program memory where $|CI| = 25.9KB$ (*multi-hop oscilloscope*). We further assume single LAT entries to be coded using 24 bits, i.e. $|LAT_h| = \#Entries(LAT_h) \cdot 3$ bytes for the proposed block size $s_h = 512$ bytes. The LAT_h then occupies 153 bytes. Compression results for the LAT_h of our benchmark applications are listed in Table I. For this setting and by applying our countermeasure an attacker's absolute gain of free program memory to upload a bogus code image $\widetilde{CI}$ would shrink below 5 bytes approximately[6].

Table 1: Maximum sizes of bogus code images $|\widetilde{CI}|$ for $s_h = 512$ bytes and various applications.

	Multi-hop oscilloscope [byte]	*BaseStation* [byte]	*Sense* [byte]
$\|CI\|$	25906	15240	2860
$\|LAT_h\|$	153	90	18
$\|PZIP(LAT_h)\|$	148	92	30
$\|PPMZ(LAT_h)\|$	163	109	48
$\|Deflate(LAT_h)\|$	181	123	48
$\|ZPAQ(LAT_h)\|$	242	188	131
max. $\|\widetilde{CI}\|$:			
1. our approach	5	0	0
2. Refs. [YWZ+07], [SMK+05]	16948	7029	1124

Again, with a larger choice of the block size s_h one could reduce the free memory space for an attacker even more. Furthermore, in case the CI is smaller also the LAT_h shrinks. E.g., if CI is the *BaseStation* respectively *Sense* application and the block size again is $s_h = 512$ bytes, the attacker will not gain free memory by compressing the LAT_h of size 90 respectively 18 bytes using the compression algorithms mentioned in this paper. Finally, the attacker could overwrite the LAT_h within the program memory for his own bogus code. In equivalence to the other program memory containing compressed code and PRW this attack is detected by the computation and subsequent verification of $x = h(nonce||C_h(CI)||LAT_h||PRW)$.

Attacks using External Memory. As shown, our proposed attestation protocol raises the attacker's overhead for generating free space within the program memory by orders of magnitude. Therefore, the attacker can not simply use the program memory to store his bogus code image $\widetilde{CI}$ in an undetected manner. A sensor node's external memory is usually multiple times slower than its program memory. Nevertheless, we have to choose

[6]The attacker can choose other compression algorithms not mentioned in this paper as well. Although unlikely, other algorithms could provide slightly better compression ratios.

the device-dependent threshold T_{em} sufficiently harsh to defend against attacks that make additional use of the external memory. Therefore, there may be cases in which a false negative will be the result of a single code attestation run. As a consequence, in case of a false negative one should repeat the code attestation protocol n times, where n is factor two the number of protocol runs in which the received x does not match the computation at the verifier. To restrict the number of iterations for the code attestation protocol we recommend to stop the protocol in case the received response x (each time with a different $nonce$) has been presented two times.

Other Attacks: DoS. The protocol is not resistant against DoS attacks. An attacker can always enforce the code attestation protocol to stop. In such situations the master device considers the code image running on the prover device as bogus.

8 Conclusions and Open Issues

The work at hand presents a code attestation protocol which in particular detects compression attacks aiming to run bogus code in an undetected manner. The code image is loaded in a compressed manner. Only LAT and PRW are loaded uncompressed. The presented approach is work in progress. Surely, more elaborated analysis are required on a proper choice of parameters like s_h, T_{pm}, n and whether or not an attacker is able to effectively compress $C_h(CI)$ without using block based compression schemes. Also the role of the cache needs to be evaluated more in depth with respect to potential security weaknesses.

9 Acknowledgments

The authors are most grateful to Aurelien Francillon and Claude Castelluccia who gave insightful comments on their related work. The work presented in this paper was supported in part by the European Commission within the STREP WSAN4CIP of the EU Framework Programme 7 for Research and Development as well as the German BMB+F SKIMS project. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the WSAN4CIP project, the SKIMS project or the European Commission.

Bibliography

[ABE+05] Abadi, M.; Budiu, M.; Erlingsson, Ú.; Ligatti, J.: Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 340–353, New York, NY, USA, ACM, 2005.

[ANN09] AbuHmed, T.; Nyamaa, N.; Nyang, D.: Software-based remote code attestation in

wireless sensor network. In *Proceedings of the 28th IEEE conference on Global telecommunications*, GLOBECOM'09, pages 4680–4687, Piscataway, NJ, USA, IEEE Press, 2009.

[CFP+09] Castelluccia, C.; Francillon, A.; Perito, D.; Soriente, C.: On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 400–409, New York, NY, USA, ACM, 2009.

[FGS09] Ferguson, C.; Gu, Q.; Shi, H.: Self-healing control flow protection in sensor applications. In *Proceedings of the second ACM conference on Wireless network security*, WiSec '09, pages 213–224, New York, NY, USA, ACM, 2009.

[Hu52] Huffman, D. A.: A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098 –1101, 1952.

[SLP+06] Seshadri, A.; Luk, M.; Perrig, A.; Doorn, L. v.; Khosla, P.: SCUBA: Secure Code Update By Attestation in sensor networks. In *Proceedings of the 5th ACM workshop on Wireless security*, WiSe '06, pages 85–94, New York, NY, USA, ACM, 2006.

[SMK+05] Shaneck, M.; Mahadevan, K.; Kher, V.; Kim, Y.: Remote Software-Based Attestation for Wireless Sensors. In Refik Molva, Gene Tsudik, and Dirk Westhoff, editors, *Security and Privacy in Ad-hoc and Sensor Networks*, volume 3813 of *Lecture Notes in Computer Science*, pages 27–41. 10.1007/11601494_3. Springer Berlin / Heidelberg, 2005.

[SPD+04] Seshadri, A.; Perrig, A.; Doorn, L. v.; Khosla, P.: SWATT: softWare-based attestation for embedded devices. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 272 – 282, May 2004.

[WC92] Wolfe, A.; Chanin, A.: Executing compressed programs on an embedded RISC architecture. *SIGMICRO Newsl.*, 23:81–91, December 1992.

[XJ03] Xu, X.; Jones, S.: Code compression for the embedded ARM/THUMB processor. In *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2003. Proceedings of the Second IEEE International Workshop on*, pages 31 –35, 2003.

[YFN+06] Yamada, H.; Fuji, D.; Nakatsuka, Y.; Hotta, T.; Shimamura, K.; Inuduka, T.; Yamazaki, T.: Micro-controller for reading out compressed instruction code and program memory for compressing instruction code and storing therein, January 2006.

[YWZ+07] Yang, Y.; Wang, X.; Zhu, S.; Cao, G.: Distributed Software-based Attestation for Node Compromise Detection in Sensor Networks. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, SRDS '07, pages 219–230, Washington, DC, USA, IEEE Computer Society, 2007.