Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Verwandte Arbeiten

Benjamin Vetter

Code Attestation mit komprimiertem, ausführbarem Code

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

Drahtlose Sensorknoten sind Systeme geringer Größe, die über Sensoren verschiedene Eigenschaften der Umgebung messen und in Form von mobilen Ad-Hoc Netzwerken mit anderen Sensorknoten oder Geräten kommunizieren. Dabei sind drahtlose Sensorknoten aufgrund ihres geringen Sicherheitsniveaus, der geringen Leistung und dem steigenden Verbreitungsgrad zahlreichen Bedrohungen ausgesetzt. Um die Vertrauenswürdigkeit der Sensorknoten zu verifizieren, kann eine Code Attestation mithilfe eines Challenge-Response Protokolls durchgeführt werden. Hierbei stellt eine vertrauenswürdige Instanz, eine Basisstation, eine Frage (Challenge) an einen Knoten. Der Knoten berechnet eine Antwort (Response) und schickt sie an die Basisstation zurück. Falls die Response korrekt ist, gilt ein Knoten weiterhin als vertrauenswürdig. Falls die Response jedoch falsch ist, wird bspw. ein Alarm ausgelöst. Die Response wird von einem Knoten mithilfe einer Hash- oder Checksummenfunktion berechnet, in die u.a. der Inhalt des Program Memories des Knotens einfließt. Um einen Knoten zu kompromittieren muss ein Angreifer idR. das Program Memory modifizieren. Daher kann nur ein unkompromittierter, vertrauenswürdiger Knoten eine korrekte Response berechnen. Jedoch haben Castelluccia u. a. (2009) gezeigt, dass die existierenden, ausschließlich software-basierten Code Attestation Protokolle von Seshadri u. a. (2004), Yang u. a. (2007), Seshadri u. a. (2006) und Shaneck u. a. (2005) erfolgreich angegriffen werden können. Bspw. kann ein Angreifer mithilfe eines sog. Kompressionsangriffs freien Speicherplatz im Program Memory eines Knotens erzeugen, indem das existierende Code Image im Program Memory des Knotens mithilfe eines verlustfreien Kompressionsverfahrens, wie bpsw. Canoncial Huffman Encoding, komprimiert wird. Anschließend kann der Angreifer Schadcode in den gewonnenen Speicherplatz einschleusen. Unser Ziel ist daher, ein sicheres Code Attestation Protokoll zu konzipieren, das insbesondere immun gegen Kompressionsangriffe ist. Hierzu laden wir ein bereits komprimiertes Code Image während des Deployments auf einen Knoten. Komprimierten Code nochmals gewinnbringend zu komprimieren birgt mehr Aufwand für einen Angreifer als unkomprimierten Code gewinnbringend zu komprimieren. Wir beabsichtigen daher den Overhead eines Angreifers durch eine geeignete Kompression soweit zu erhöhen, dass er sich nicht mehr innerhalb möglichst groß definierbarer zeitlicher Schranken aufhalten kann. Um komprimierten Code auf einem Sensorknoten auszuführen benötigt unser Verfahren einen Microcontroller, der den komprimierten Code zur Laufzeit dekomprimiert. Daher müssen die verwendeten Senorknoten vor dem Deployment mit einem entsprechenden Microcontroller bestückt werden.

Diese Arbeit identifiziert die für unser Protokoll relevanten Arbeiten und Forschungsgruppen, fasst die Arbeiten zusammen und stellt sie gegenüber. Unser Protokoll wird in den Kontext der Arbeiten eingeordnet und von den Arbeiten abgegrenzt. Kapitel 2.1 stellt die relevanten Arbeiten bzgl. Code Attestation Protokollen vor und Kapitel 2.2 identifiziert die Arbeiten bzgl. der Dekompression zur Laufzeit mit Fokus auf Sensorknoten. Kapitel 3.1 ordnet unser Protokoll in die relevanten Arbeiten ein und Kapitel 3.2 zeigt, welche Aspekte der Arbeiten wir explizit nicht in unserem Protokoll einsetzen, d.h. grenzt unser Protokoll ab. Abschließend fasst Kapitel 4 die Ergebnisse dieser Arbeit zusammen.

2 Verwandte Arbeiten

2.1 Code Attestation Protokolle

Die Arbeiten bzgl. Code Attestation Protokollen gliedern sich in zwei Themengebiete. Einerseits konzipieren die Arbeiten neue Protokolle und andererseits greift eine Arbeit die Sicherheit vorhandener Protokolle an. Dieses Kapitel stellt die vier Arbeiten von Seshadri u. a. (2004), Yang u. a. (2007), Seshadri u. a. (2006) und Shaneck u. a. (2005) vor, die neue Protokolle konzipieren und von drei unterschiedlichen Forschungsgruppen stammen. Zusätzlich zeigt dieses Kapitel für jedes vorgestellte Protokoll, wie die Ergebnisse von Castelluccia u. a. (2009) die Sicherheit des jeweiligen Protokolls widerlegen. Die vier Protokolle basieren auf der grundlegenden Annahme, dass ein Angreifer zusätzliche Zeit benötigt um vor einem Attestation Algorithmus zu verstecken, dass ein Knoten kompromittiert ist. Ein kompromittierter Knoten benötigt dabei mehr Zeit um eine korrekte Response zu berechenen als ein unkompromittierter Knoten. Anhand dieses Overheads versuchen die Protokolle einen Angreifer zu entdecken. Sensorknoten basieren idR. auf einer Harvard Architektur. Code ist auf Harvard Architekturen nur aus dem Program Memory heraus ausführbar. Daher gehen die vier Protokolle zusätzlich davon aus, dass ein Angreifer freien Speicherplatz im Program Memory eines Knotens benötigt.

SWATT. Das Protokoll SWATT, SoftWare-based ATTestation for Embedded Devices, stammt von Seshadri u. a. (2004). Die verantwortliche Forschungsgruppe besteht aus Arvind Seshadri, Adrian Perrig und Pradeep Khosla von CMU/CyLab, sowie Leendert van Doorn von IBM. Wie soeben gezeigt, versucht ein Angreifer Schadcode in das Program Memory eines Knotens einzuschleusen um einen Knoten zu kompromittieren. Um Änderungen des Program

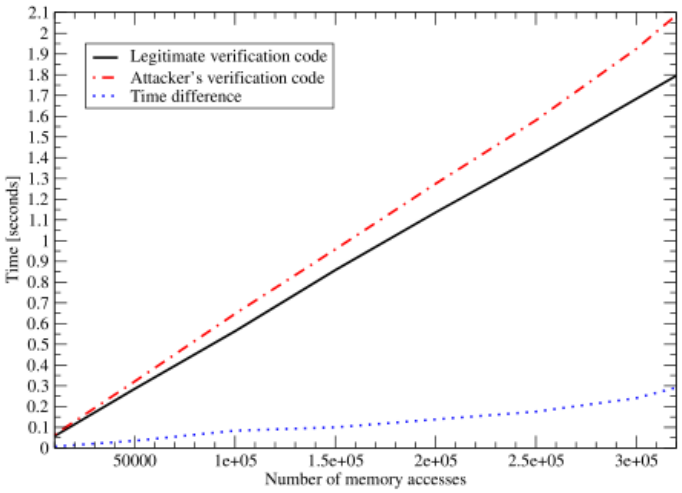

Abbildung 2.1: Abweichender Overhead (blau) eines Angreifers (rot) durch kumulierendes Test and Redirect (Seshadri u. a., 2004).

Memories zu entdecken, berechnet SWATT eine Checksumme, in die der Inhalt des Program Memories einfließt. Daher muss ein Angreifer die ursprünglichen Daten im Program Memory an einen anderen Ort kopieren, bevor er die Daten mit Schadcode überschreibt. Andernfalls ist der Angreifer während einer Code Attestation nicht in der Lage eine korrekte Response zu berechnen. SWATT schlägt das sogenannte Pseudorandom Memory Traversal vor, um den Overhead eines Angreifers soweit zu erhöhen, dass er enttarnt werden kann. Hierbei wird ein Pseudozufallszahlengenerator (PRNG) mit der empfangenen Challenge als Startwert initialisiert. Der Attestation Algorithmus von SWATT iteriert über die Adressen aller Bytes des Program Memories, wobei die Reihenfolge der Byteadressen durch die Zufallszahlenfolge des PRNG festgelegt ist. Hierdurch muss ein Angreifer während der Berechnung der Response für jede Speicherstelle des Pseudorandom Memory Traversals überprüfen, ob die Speicherstelle mit Schadcode überschrieben wurde oder nicht und ggf. durch einen Zugriff auf die Kopie ersetzen. Ein Angreifer muss bspw. ein zusätzliches $if()$, von Castelluccia u. a. (2009) als Test and Redirect bezeichnet, in jeder Iteration ausführen. Daher muss ein Angreifer eine zusätzliche Menge von Instruktionen ausführen. Je mehr Speicherstellen in die Checksumme fließen, desto mehr Overhead resultiert für einen Angreifer, wie in Abbildung 2.1 zu sehen ist. Daher kann der zeitliche Overhead eines Angreifers durch die Anzahl an Iterationen soweit erhöht werden, dass er enttarnt wird.

Angriff auf SWATT. Castelluccia u. a. (2009) widerlegen die Sicherheit des Protokolls SWATT. Die verantwortliche Forschungsgruppe besteht aus Claude Castelluccia und Aurélien Francillon von INRIA, sowie Claudio Soriente von der Universtität Kalifornien, Irvine. Die Autoren von SWATT gehen davon aus, dass ihr eigens entwickelter Angriff auf SWATT durch Test and Redirect bereits optimal effizient ist und die minimale Anzahl an Instruktionen auf-

weist. Die Anzahl an Instruktionen, die für einen Angriff notwendig sind, legt den zeitlichen Overhead eines Angreifers fest. Um die Sicherheit von SWATT zu widerlegen und unterhalb der erkennbaren zeitlichen Schranke zu bleiben, reicht es aus, die für einen Angriff notwendige Anzahl an Instruktionen zu reduzieren. Die Autoren von SWATT gehen von drei Instruktionen für einen Test and Redirect aus. Im Gegensatz dazu benötigen Castelluccia u. a. (2009) nur zwei Instruktionen für einen Test and Redirect. Der Overhead eines Angriffs ist daher geringer als die Autoren von SWATT annehmen. Sie schlussfolgern, dass es für Entwickler von Protokollen äußerst schwierig zu zeigen sei, was der optimal effiziente Angriff auf ein Protokoll ist. Daher könne nicht ausgeschlossen werden, dass noch effizientere Angriffe auf SWATT existieren, die nicht anhand eines zeitlichen Overheads entdeckt werden können.

SCUBA. Das Protokoll SCUBA, Secure Code Update by Attestation in Sensor Networks, stammt von Seshadri u. a. (2006). Bei der Forschungsgruppe handelt es sich um die selbe Gruppe, die das Protokoll SWATT entwickelt hat, erweitert um Mark Luk von CMU/CyLab. SCUBA basiert auf dem Mechanismus der Indisputable Code Execution (ICE). Hierbei garantiert der ICE-Mechanismus, dass eine Menge Code auf einem Knoten ausgeführt wird, ohne dass anderer Code des Knotens die Ausführung unbemerkt beeinflussen kann. Nachdem der Knoten eine Challenge emfangen hat, berechnet die ICE-Funktion zunächst eine Checksumme von sich selbst, indem bspw. die auftretenden Werte des Program Counters in die Checksumme fließen. Hierdurch stellt die ICE-Funktion sicher, dass sie selbst unmodifiziert vorliegt, da andernfalls die Werte des Program Counters variieren. Anschließend berechnet die ICE-Funktion eine Checksumme des Codes, der ausgeführt werden soll. Hierdurch stellt die ICE-Funktion sicher, dass auch der auszuführende Code unmodifiziert vorliegt. Zuletzt deaktiviert die ICE-Funktion die Interrupts des Knotens und führt den gewünschten Code aus. Die ICE-Funktion deaktiviert die Interrupts, damit kein evtl. auftrender Interrupt den auszuführende Code unterbrechen und manipulieren kann. Mithilfe des Vertrauensankers der ICE-Funktion kann bspw. ein Angreifer entdeckt werden, der einen modifizierten Code Attestation Algorithmus auf einen Knoten läd, da die von der ICE-Funktion berechnete Checksumme dann falsch ist.

Angriff auf SCUBA. Castelluccia u. a. (2009) widerlegen allerdings auch die Sicherheit des SCUBA Protokolls, indem sie eine zuvor unbekannte Schwachstelle ausnutzen. Dabei kippen sie das Most Significant Bit zweier Eingabeparameter von SCUBA, ohne dass sich das Ergebnis der Checksumme von SCUBA ändert. Bspw. kann ein Angreifer das Most Significant Bit des Program Counters und in jedem Byte des zu überprüfenden Speicherinhalts kippen, ohne dass sich das Ergebnis der Checksumme ändert. Die Änderungen heben sich durch das exklusive Oder bei der Berechnung von $PC \oplus mem[current\_address]$ von SCUBA auf. Dieses Vorgehen entspricht einem Angreifer, der eine exakte Kopie der ICE-Funktion an einer anderen Adresse im Program Memory ausführt. Die zu verwendende Adresse berechnet sich durch das Kippen des Most Significant Bits des Program Counters. Die kopierte

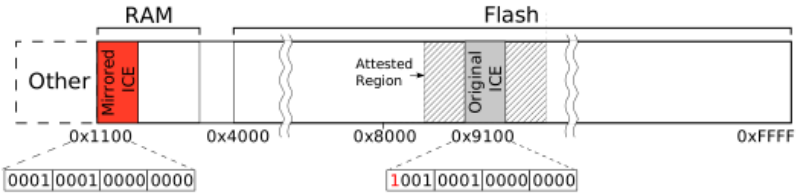

Abbildung 2.2: Angriff auf SCUBA indem eine Kopie der ICE-Funktion ausgeführt wird (Castelluccia u. a., 2009).

ICE-Funktion kann die Kontrolle anschließend unbemerkt an beliebigen Code des Angreifers übergeben, da der Angreifer neben der Kopie der ICE-Funktion beliebigen Schadcode platzieren kann, der nicht von der Code Attestation bei SCUBA berücksichtigt wird. Der Schadcode wird anstelle des Codes ausgeführt, für den bereits eine korrekte Checksumme, d.i. eine korrekte Response berechnet wurde. Dieser Angriff ist in Abbildung 2.2 dargestellt.

Noise Filling. Yang u. a. (2007) schlagen ein weiteres Code Attestation Protokoll vor. Die verantwortliche Forschungsgruppe besteht aus Yi Yang, Xinran Wang, Sencun Zhu und Guohong Cao von der Pennsylvania State Universität. Das Code Image im Program Memory eines Sensorknotens ist idR. wesentlich kleiner als das gesamte Program Memory von bspw. 128 Kilobytes (Avr, 2011). Der ungenutzte Speicher ist idR. mit einem konstanten Wert wie bspw. 0xFF gefüllt. Daher kann ein Angreifer Schadcode in den ungenutzten Speicher einschleusen und anschließend dennoch die Code Attestation bestehen. Während der Code Attestation verwendet der Angreifer dazu das originale Code Image, das noch im Program Memory vorhanden ist und berechnet eine korrekte Response. Yang u. a. (2007) schlagen daher vor, den ungenutzten Speicher mit Zufallsdaten statt eines konstanten Werts zu beschreiben. Die Zufallsdaten sind der Basisstation bekannt und fließen in die Checksumme der Response mit ein. Daher kann ein Angreifer die Zufallsdaten nicht mit Schadcode überschreiben, da er andernfalls keine korrekte Response berechnen kann. Dieses Protokoll überlässt einem Angreifer keinen freien Speicher im Program Memory.

Kompressionsangriff. Castelluccia u. a. (2009) widerlegen auch die Sicherheit des Noise Filling Verfahrens, indem sie freien Speicherplatz im Program Memory eines Knotens durch einen sog. Kompressionsangriff erzeugen. Hierzu komprimiert ein Angreifer das Code Image im Program Memory eines Knotens mit einem verlustfreien Kompressionsverfahren, wie bspw. Canonical Huffman Encoding. Im so gewonnen Speicherplatz kann der Angreifer abermals Schadcode einschleusen und dennoch die Code Attestation bestehen. Hierzu dekomprimiert der Angreifer während einer Attestation das komprimierte Code Image und erhält somit alle notwendigen Eingabeparameter um eine korrekte Response zu berechnen.

Obfuskierung. Shaneck u. a. (2005) schlagen vor, den Code Attestation Algorithmus erst zu Beginn einer Attestation auf einen Knoten zu übertragen. Die verantwortliche Forschungsgruppe besteht aus Mark Shaneck, Karthikeyan Mahadevan, Vishal Kher und Yongdae Kim

von der Universität Minnesota. Ein Angreifer muss hierbei den Algorithmus zunächst analysieren, um Schadcode vor dem unbekannten Algorithmus zu verstecken. Der Angreifer benötigt daher zusätzliche Zeit um eine korrekte Response zu berechnen. Anhand dieses Overheads versucht das Protokoll den Angreifer zu entdecken. Um die Analyse des Angreifers zu erschweren und den Overhead zu erhöhen, verwendet das Protokoll Methoden der Obfuskierung. Bei jeder Attestation wird ein unterschiedlicher Algorithmus verwendet und der Self Modifying Code des Algorithmus modifiziert sich während der Ausführung selbst, was eine Analyse durch den Angreifer erschwert.

Probleme der Obfuskierung. Laut Castelluccia u. a. (2009) ist es jedoch ausgesprochen kompliziert, Self Modifying Code zu entwickeln und zu implementieren, gerade weil Self Modifying Code nur schwer zu verstehen und zu analysieren ist. Insbesondere auf Harvard Architekturen ist Self Modifying Code nicht oder nur ineffizient implementierbar. Daher halten sie den Einsatz von Self Modifying Code in sicherheitsrelevanten Prokollen, wie der Code Attestation, für eine fragwürdige Wahl.

2.2 Dekompressionsverfahren

Sensorknoten müssen in unserem Protokoll in der Lage sein, komprimierten Code zur Laufzeit zu dekomprimieren. Daher werden nun die drei diesbzgl. verwandten Arbeiten von Wolfe und Chanin (1992), Xu und Jones (2003) und Lefurgy u. a. (1997), die von drei unterschiedlichen Forschungsgruppen stammen, vorgestellt. Die Arbeiten untergliedern sich in zwei Kategorien. Erstens existieren Arbeiten zu blockbasierten Verfahren, bei denen Codeblöcke fester Länge vor dem Deployment komprimiert und zur Laufzeit dekomprimiert werden. Zweitens existieren dictionarybasierte Verfahren, die häufig im Code vorkommende Codesequenzen in einem Dictionary speichern.

Blockbasierte Kompression. Wolfe und Chanin (1992) schlagen vor, ein Code Image in Codeblöcke fester Länge zu zerlegen und einzeln zu komprimieren. Die verantwortliche Forschungsgruppe besteht aus Andrew Wolfe und Alex Chanin von der Universität Princeton. Da die komprimierten Codeblöcke nicht mehr gleich lang sind, wird eine Datenstruktur benötigt, die die Offsets der komprimierten Blöcke bereithält. Diese Datenstruktur bezeichnen die Autoren als Line Address Table (LAT). Wie in Abbildung 2.3 zu sehen ist, verwenden die Autoren einen Cache, um den jeweils aktuell ausgeführten, dekomprimierten Block zwischenzuspeichern. Ein Cache Miss tritt auf, wenn bspw. eine jmp Instruktion zu einer Adresse ausgeführt wird, die aktuell nicht im Cache zwischengespeichert ist. Der Microcontroller, in Abbildung 2.3 als Cache Refill Engine bezeichnet, dekomprimiert im Falle eines Cache Miss den Block in den gesprungen werden soll anhand der Offsets der LAT und füllt den Cache mit den dekomprimierten Daten. Anschließend kann das Programm fortfahren. Wolfe und Chanin (1992) schlagen als Cache- und Blockgröße 32 Bytes vor. Dabei erreichen sie

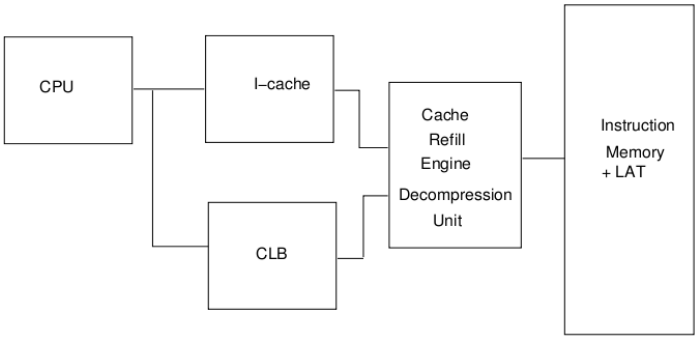

Abbildung 2.3: Blockbasierte Dekompression zur Laufzeit mit LAT, Cache und Microcontroller (Wolfe und Chanin, 1992).

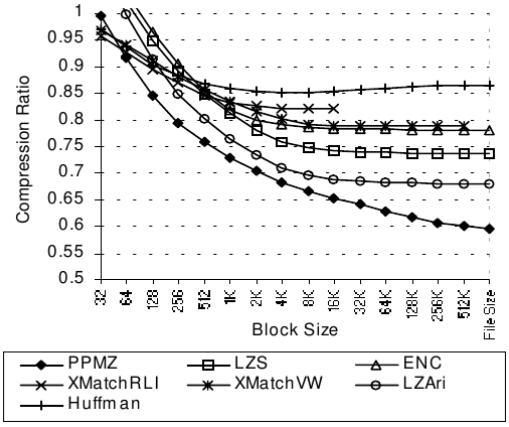

Abbildung 2.4: Kompressionsraten für unterschiedliche Blockgrößen bei 16-bit THUMB Code[2](Xu und Jones, 2003).

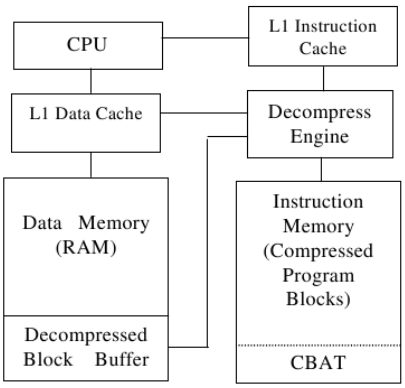

Abbildung 2.5: Architektur mit Cache innerhalb des RAM (Xu und Jones, 2003).

laut Bonny und Henkel (2010) Kompressionsraten von ca. 73% für MIPS Code[1]. Die Ausführungsgeschwindigkeiten reduzieren sich bei diesem Verfahren idR. um weniger als 10%.

Cache innerhalb des RAM. Wie in Abbildung 2.4 zu sehen ist, führen größere Blockgrößen zu besseren Kompressionsraten. Vorhandene Caches eines Systems, wie L1 oder L2 Caches, sind jedoch zu klein um größere Blockgrößen zu unterstützen. Daher schlagen Xianhong Xu und Simon Jones von der Universität Bath vor, wie in Abbildung 2.5 gezeigt, einen Teil des RAM als Decompressed Block Buffer (Cache) zu verwenden (Xu und Jones, 2003). Hierdurch können deutlich größere Blockgrößen verwendet werden. Die Compressed Block

[1] Ausgehend von einer Originalgröße von 100% werden also 27% eingespart.
[2] Laut Xu und Jones (2003) zeichnet sich 16-bit THUMB Code von ARM durch eine höhere Dichte aus. Hohe Kompressionsraten sind daher schwieriger zu erzielen.

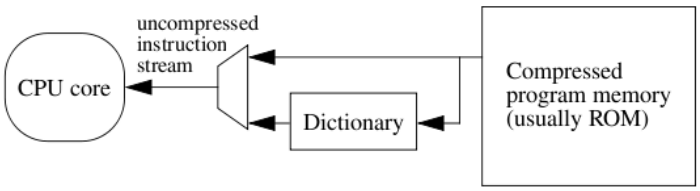

Abbildung 2.6: Dekompression von Lefurgy u. a. (1997) mithilfe eines Dictionaries.

Address Table (CBAT) entspricht der LAT. Der Instruction Cache entspricht dem von Wolfe und Chanin (1992) verwendeten Cache und bildet gemeinsam mit dem Cache innerhalb des RAM eine Cachehierarchie. Der Data Cache ist notwendig, da innerhalb des Programmcodes bspw. Konstanten definiert werden. Zugriffe auf diese Konstanten führen ggf. zu Cache Misses, da die Konstanten idR. nicht im aktuellen Codeblock definiert sind. Daher reduziert der zusätzliche Data Cache die Anzahl an Cache Misses und verbessert die Performance. Je größer die Blockgröße bei diesem Verfahren gewählt wird, desto seltener treten Cache Misses auf. Andererseits wird bei größeren Blockgrößen auch mehr Zeit benötigt, um einen Block zu dekompromieren und den Cache neu aufzufüllen. Die Kompressionsrate des Verfahrens hängt stark vom verwendeten Kompressionsverfahren und der Blockgröße ab. Für Blockgrößen von 256 Bytes bis zwei Kilobytes und 16-bit THUMB Code[3] werden Kompressionsraten von 90% bis 70% erzielt (vgl. Abbildung 2.4).

Dictionarybasierte Kompression. Lefurgy u. a. (1997) schlagen ein dictionarybasiertes Verfahren vor um komprimierten Code zur Laufzeit zu dekomprimieren. Die verantwortliche Forschungsgruppe besteht aus Charles Lefurgy, Peter Bird, I-Cheng Chen und Trevor Mudge von der Universität Michigan. Das Verfahren identifiziert häufig im Code vorkommende Codesequenzen und speichert die Sequenzen in einem Dictionary. Im Programmcode werden die Codesequenzen durch kürzere Codewörter ersetzt, die als Index ins Dictionary fungieren und die Codegröße reduzieren, d.h. die Kompression erzielen. Abbildung 2.6 zeigt die verwendete Architektur bzgl. der Dekompression. Ein Microcontroller dekomprimiert den komprimierten Stream auf dem Weg zur CPU mithilfe des Dictionaries. Die Dekompression ist hierbei sehr performant, da der Microcontroller die Codesequenzen anhand des Index effizient im Dictionary nachschlagen kann. Die Kompressionsraten des Verfahrens liegen bei maximal 70% bis 50% für Code der PowerPC Architektur.

[3] 16-bit THUMB Code von ARM zeichnet sich durch eine höhere Dichte aus. Hohe Kompressionsraten sind daher schwieriger zu erzielen (Xu und Jones, 2003).

3 Einordnung und Abgrenzung

3.1 Einordnung

Wir kombinieren Mechanismen der vorgestellten Protokolle. Unser Protokoll ist ein Challenge-Response basiertes Protokoll, wie ursprünglich von Seshadri u. a. (2004) vorgeschlagen. Nur ein unkomprimierter Knoten soll in der Lage sein, eine korrekte Response innerhalb eines definierten Zeitraums zu berechnen. Um den Overhead des Angreifers in die Höhe zu treiben verwenden wir den Mechanismus des Pseudorandom Memory Traversals, ebenfalls von Seshadri u. a. (2004) vorgeschlagen, sodass die Bytes des Program Memories in pseudozufälliger Reihenfolge in die Hash- oder Checksummenfunktion fließen. Wir laden jedoch ein blockweise komprimiertes Code Image auf einen Knoten. Daher muss ein Angreifer das bereits komprimierte Code Image nochmals besser komprimieren um freien Speicherplatz zu erlangen[1]. Wie von uns gezeigt, muss der Angreifer dann während der Code Attestation jedoch jeden von ihm komprimierten bspw. 1024 Bytes großen Block 1024 mal dekomprimieren um an jedes einzelne Byte zu gelangen, das in die Hash- oder Checksummenfunktion fließt (Vetter und Westhoff, 2011). Daher entwickeln wir das Pseudorandom Memory Traversal konzeptionell zu einem Pseudorandom Memory-Block Traversal weiter und erhöhen den Overhead eines Angreifers dadurch, im Gegensatz zu SWATT, um Größenordnungen. U.a. da wir das Code Image komprimiert auf einen Knoten laden, verbleibt jedoch freier Platz im Program Memory in den ein Angreifer Schadcode einschleusen kann. Daher verwenden wir das Noise Filling Verfahren von Yang u. a. (2007), sodass wir den freien Speicherplatz mit Zufallsdaten beschreiben und ein Angreifer nicht trivialerweise Schadcode in freien Speicherplatz einschleusen kann.

Da wir das Code Image blockweise komprimieren, verwenden wir eine blockweise Kompression, bei der ein Cache, eine LAT und ein Microcontroller verwendet wird. Die kleinen Blockgrößen von 32 Bytes erlauben jedoch nur für die von Wolfe und Chanin (1992) verwendete Huffmankompression von MIPS Code Kompressionsraten von bis zu 73%. Das gilt nicht für beliebige andere Kompressionsverfahren und Architekturen. Unser Protokoll soll

[1] Der Angreifer muss bessere Kompressionsraten erzielen. Hierzu muss er eine größere Blockgröße oder ein besseres Kompressionsverfahren wählen als unser Protokoll (Vetter und Westhoff, 2011).

flexibel die Kompressionsverfahren wechseln können, bspw. falls zukünftige Kompressionsverfahren bessere Kompressionsraten erzielen. Durch eine blockweise Kompression mit größeren Blockgrößen können die Blöcke mit beliebigen Kompressionsverfahren, d.h. mit statistischen oder dictionarybasierten Kompressionsverfahren, komprimiert werden und dennoch gute Kompressionsraten erzielen (Xu und Jones, 2003). Daher verwenden wir einen Teil des RAM als Cache um größere Caches und größere Blockgrößen zu erzielen.

3.2 Abgrenzung

Unser Protokoll ist unseres Wissens nach das erste Code Attestation Protokoll, das eine Kompression als Sicherheitsmechanismus verwendet. Dementsprechend ist unser Protokoll das Einzige, das ein komprimiertes Code Image auf einen Knoten läd und zur Laufzeit dekomprimiert. Ein Kompressionsangriff, wie von Castelluccia u. a. (2009) auf das Protokoll von Yang u. a. (2007), ist daher grundsätzlich schwieriger. Im Gegensatz zu den vorgestellten Protokollen benötigt unser Protokoll einen Micrcontroller. Die Hardwareunterstützung hat, neben positiven Sicherheitseigenschaften und der effektiveren Nutzung des Program Memories, auch negative Auswirkungen. Die Flexibilität unseres Protokolls nimmt ab und die Kosten eines Knotens steigen. Angesichts der Angriffe auf die ausschließlich softwarebasierten Protokolle ist es jedoch fraglich, ob ein sicheres Protokoll ohne Hardwareunterstützung überhaupt möglich ist. Wie jedes andere Protokoll, entdecken wir einen Angreifer anhand eines zeitlichen Overheads. Ein kompromittierter Knoten benötigt dabei mehr Zeit um eine korrekte Response zu berechnen. Wir erhöhen jedoch den Overhead eines Angreifers durch die Kombination von Pseudorandom Memory Traversal und blockweiser Kompression um Größenordnungen und können dadurch insbesondere Kompressionangriffe entdecken (Vetter und Westhoff, 2011). Demgegenüber kann bspw. das Protokoll SWATT den Overhead nur geringfügig erhöhen (vgl. Abbildung 2.1). Die Kritik bzgl. Protokollen, die auf sehr engen zeitlichen Schranken basieren, gelten daher für unser Protokoll nicht gleichermaßen. Ein offenes Problem unseres Protokolls ist jedoch, dass ein Angreifer ggf. Code in den zusätzlichen Cache einschleusen und einen Knoten dadurch ggf. unbemerkt kompromittieren kann. Dieses spezifische Problem unseres Protokolls müssen wir noch lösen. Es ist äußerst schwierig, ein sicheres Protokoll zu entwickeln, das auf Indisputable Code Execution (ICE) basiert. Außerdem sind die Werte des Program Counters nicht auf allen Architekturen verfügbar. Daher sehen wir davon ab, ICE als Sicherheitsmechanismus zu verwenden. Die Code Attestation von Shaneck u. a. (2005) mithilfe von Obfuskierung basiert auf Self Modifying Code. Die Entwicklung von Self Modifying Code ist jedoch schwierig und Self Modifying Code ist auf Harvard Architekturen nicht möglich, da Code auf Harvard Architekturen idR. nicht

das Program Memory modifizieren kann[2]. Daher sollten sicherheitskritische Mechanismen auch unserer Ansicht nach nicht auf Self Modifying Code beruhen.

Laut Bell u. a. (1990) existiert für jedes dictionarybasierte Verfahren nachweislich ein mindestens ebenso gutes statistisches Kompressionsverfahren. Daher beabsichtigen wir, keine dictionarybasierten Kompressionsverfahren zu verwenden. Falls widererwartend dictionarybasierte Verfahren eingesetzt werden, dann nur kombiniert mit einer blockweisen Kompression und um die jeweiligen Blöcke zu komprimieren. Von dictionarybasierten Verfahren, wie bspw. von Lefurgy u. a. (1997), sehen wir jedoch ab. Die Kompressionsraten der vorgestellten Kompressionsverfahren sind u.a. sehr abhängig von der verwendeten Architektur. Daher sind die Kompressionsraten nur begrenzt vergleichbar und zusätzliche Analysen sind notwendig. Unser Protokoll benötigt den beschriebenen Microcontroller als zusätzliche Hardwarekomponente. Da z.Zt. kein derartiger Microcontroller für Sensorknoten verfügbar ist, müssen wir den Microcontroller für unsere Proof of Concept Implementierung emulieren. Die Emulation des Microcontrollers kann bspw. auf Basis des quelloffenen Simulators Avrora (2011) erfolgen. Ob zukünftig ein Microcontroller in Hardware verfügbar sein wird ist jedoch ungewiss.

4 Zusammenfassung

Castelluccia u. a. (2009) haben gezeigt, dass die Entwicklung eines sicheren Code Attestation Protokolls schwierig und fehleranfällig ist. Die bisher vorgeschlagenen Protokolle sind als unsicher anzusehen. Da Code Attestation Protokolle sicherheitskritisch sind, sollten die verwendeten Mechanismen möglichst wenig Komplexität aufweisen. Wir kombieren Ideen existierender Protokolle, wie das Noise Filling von Yang u. a. (2007) und das Pseudorandom Memory Traversal von Seshadri u. a. (2004), mit blockweisen Kompressionsverfahren von Wolfe und Chanin (1992) und Xu und Jones (2003). Dadurch entwickeln wir das Pseudorandom Memory Traversal zu einem Pseudorandom Memory Block Traversal weiter. Wir laden ausschließlich komprimierten Code auf einen Sensorknoten, den ein Microcontroller zur Laufzeit dekomprimiert. Dadurch erhöhen wir den Overhead eines Kompressionsangriffs um Größenordnungen, sind jedoch auf Hardwareunterstützung angewiesen.

[2] Nur Code des Bootloaders kann das Program Memory modifizieren (Francillon und Castelluccia, 2008).

Literaturverzeichnis

[Avr 2011] CORPORATION, Atmel: *AVR 8-bit Instruction Set*. http://www.atmel.com/atmel/acrobat/doc0856.pdf. July 24, 2011. – URL http://www.atmel.com/atmel/acrobat/doc0856.pdf

[Avrora 2011] GROUP, UCLA C.: *Avrora - The AVR Simulation and Analysis Framework*. http://compilers.cs.ucla.edu/avrora/. July 24, 2011. – URL http://compilers.cs.ucla.edu/avrora/

[Bell u. a. 1990] BELL, T.C. ; CLEARY, J.G. ; WITTEN, I.H.: *Text compression*. Prentice Hall, 1990 (Prentice Hall advanced reference series: Computer science). – URL http://books.google.com/books?id=sdZQAAAAMAAJ. – ISBN 9780139119910

[Bonny und Henkel 2010] BONNY, Talal ; HENKEL, Jörg: Huffman-based code compression techniques for embedded processors. In: *ACM Trans. Des. Autom. Electron. Syst.* 15 (2010), October, S. 31:1–31:37. – URL http://doi.acm.org/10.1145/1835420.1835424. – ISSN 1084-4309

[Castelluccia u. a. 2009] CASTELLUCCIA, Claude ; FRANCILLON, Aurélien ; PERITO, Daniele ; SORIENTE, Claudio: On the difficulty of software-based attestation of embedded devices. In: *Proceedings of the 16th ACM conference on Computer and communications security*. New York, NY, USA : ACM, 2009 (CCS '09), S. 400–409. – URL http://doi.acm.org/10.1145/1653662.1653711. – ISBN 978-1-60558-894-0

[Francillon und Castelluccia 2008] FRANCILLON, Aurélien ; CASTELLUCCIA, Claude: Code injection attacks on harvard-architecture devices. In: *Proceedings of the 15th ACM conference on Computer and communications security*. New York, NY, USA : ACM, 2008 (CCS '08), S. 15–26. – URL http://doi.acm.org/10.1145/1455770.1455775. – ISBN 978-1-59593-810-7

[Lefurgy u. a. 1997] LEFURGY, Charles ; BIRD, Peter ; CHEN, I-Cheng ; MUDGE, Trevor: Improving code density using compression techniques. In: *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA : IEEE Computer Society, 1997 (MICRO 30), S. 194–203. – URL http://portal.acm.org/citation.cfm?id=266800.266819. – ISBN 0-8186-7977-8

[Seshadri u. a. 2004] SESHADRI, A. ; PERRIG, A. ; DOORN, L. van ; KHOSLA, P.: SWATT: softWare-based attestation for embedded devices. In: *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, Mai 2004, S. 272 – 282. – ISSN 1081-6011

[Seshadri u. a. 2006] SESHADRI, Arvind ; LUK, Mark ; PERRIG, Adrian ; DOORN, Leendert van ; KHOSLA, Pradeep: SCUBA: Secure Code Update By Attestation in sensor networks. In: *Proceedings of the 5th ACM workshop on Wireless security*. New York, NY, USA : ACM, 2006 (WiSe '06), S. 85–94. – URL http://doi.acm.org/10.1145/1161289.1161306. – ISBN 1-59593-557-6

[Shaneck u. a. 2005] SHANECK, Mark ; MAHADEVAN, Karthikeyan ; KHER, Vishal ; KIM, Yongdae: Remote Software-Based Attestation for Wireless Sensors. In: MOLVA, Refik (Hrsg.) ; TSUDIK, Gene (Hrsg.) ; WESTHOFF, Dirk (Hrsg.): *Security and Privacy in Ad-hoc and Sensor Networks* Bd. 3813. Springer Berlin / Heidelberg, 2005, S. 27–41. – URL http://dx.doi.org/10.1007/11601494_3. – 10.1007/11601494_3

[Vetter und Westhoff 2011] VETTER, Benjamin ; WESTHOFF, Dirk ; EICHLER, Gerald (Hrsg.) ; KÜPPER, Axel (Hrsg.) ; SCHAU, Volkmar (Hrsg.) ; FOUCHAL, Hacene (Hrsg.) ; UNGER, Herwig (Hrsg.): *Lecture Notes in Informatics (LNI-P)*. Bd. 186: *Code Attestation With Compressed Instruction Code. Proceeding of the 11th International Conference on Innovative Internet Community Systems I2CS 2011*. Bonn, Germany : GI-Edition, Bonner Köllen Verlag, June 2011. – URL http://www.i2cs.uni-jena.de/. – ISBN 978-3-88579-280-1

[Wolfe und Chanin 1992] WOLFE, Andrew ; CHANIN, Alex: Executing compressed programs on an embedded RISC architecture. In: *SIGMICRO Newsl.* 23 (1992), December, S. 81–91. – URL http://doi.acm.org/10.1145/144965.145003. – ISSN 1050-916X

[Xu und Jones 2003] XU, Xianhong ; 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*, 2003, S. 31 –35

[Yang u. a. 2007] YANG, Yi ; WANG, Xinran ; ZHU, Sencun ; CAO, Guohong: Distributed Software-based Attestation for Node Compromise Detection in Sensor Networks. In: *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*. Washington, DC, USA : IEEE Computer Society, 2007 (SRDS '07), S. 219–230. – URL http://portal.acm.org/citation.cfm?id=1308172.1308237. – ISBN 0-7695-2995-X