WebAssembly für die Industrie 4.0

Sichere, skalierbare Plattformen mit Bytecode-basierten Virtuellen Maschinen

Stefan Wallentowitz und Markus Friedrich, Hochschule München

WebAssembly for the Industrial Internet-of-Things

The industrial internet-of-things is built on heterogeneous hardware/software platforms, which vary in their architecture and performance. Virtualization is an important technology for building secure and portable applications on scalable IoT deployments. Bytecode based virtual machines are coming back into focus with WebAssembly, a technology that started in the web browser and now raises interest in other use cases. In this paper we discuss a vision of scalable WebAssembly deployments in the industrial internet-of-things.

Keywords:
virtualization, WebAssembly, OT security, IoT

Prof. Dr.-Ing. Stefan Wallentowitz ist Professor an der Hochschule München. Er hat langjährige Erfahrung im Bereich der Sicherheit eingebetteter Systeme, insbesondere SmartCards. Er forscht und entwickelt an Laufzeitsystemen für das Internet-der-Dinge.

Prof. Dr. Markus Friedrich ist Professor an der Hochschule München. Er ist spezialisiert im Themenbereich Visual Computing und Machine Learning. Er forscht u. a. an Lösungen zu Computer Vision Problemen auf Edge-Geräten.

stefan.wallentowitz@hm.edu
https://wasm4iot.org

Mit der zunehmenden Vernetzung steigt die Heterogenität der Plattformen in einem IoT-System. Endpunkte verschiedener Leistungsklassen haben unterschiedliche Arten von Betriebssystemen und Prozessoren, während das Gesamtsystem zusätzlich noch zentrale Server oder die Cloud umfasst. Edge-Geräte werden heute zusätzlich noch integriert, um den steigenden Leistungsanforderungen gerecht zu werden und hohe Durchsätze und niedrige Latenzen zu erreichen. Die sichere und portierbare Programmierung dieser Geräte für anspruchsvolle Aufgaben im industriellen Umfeld, wie zum Beispiel Computer Vision, ist eine Herausforderung. Bytecode-basierte Virtuelle Maschinen haben diese beiden Eigenschaften und sind mit Java seit längerer Zeit vertreten. Seit einigen Jahren strebt WebAssembly auf, das aus beliebigen Programmiersprachen übersetzt werden kann, und findet zunehmend Verbreitung über den Browser hinaus. Dieser Beitrag beschreibt die technischen Grundlagen und zeigt Möglichkeiten auf, wie WebAssembly eine skalierbare Lösung als Baustein der Operational Cybersicherheit von industriellen Anwendungen werden kann.

Die sichere Separierung von verschiedenen Software-Modulen auf Rechnersystemen ist inzwischen auch in eingebetteten Geräten ein wichtiges Thema. Nicht nur aus Gründen der funktionalen Sicherheit, sondern auch der Cybersicherheit sollen verschiedene Software-Module separiert werden, da sie zum Beispiel aus verschiedenen Quellen kommen oder die Resilienz erhöht werden soll.

Um eine strenge Isolation zu erreichen und gleichzeitig ein hohes Maß an Portabilität zu ermöglichen, hat sich das Prinzip der Virtualisierung durchgesetzt. System-Virtualisierung mit Hypervisoren teilt eine Rechnerplattform zwischen mehreren Instanzen von Betriebssystemen. Container-Virtualisierung innerhalb eines Betriebssystems hat sich im Umfeld der Cloud in Form von Docker und ähnlichen Lösungen durchgesetzt. Bei der Applikations-Virtualisierung wird nicht ein Rechner selbst, sondern ausschließlich die Laufzeitumgebung virtualisiert. Die Verwendung von Bytecode erlaubt dabei eine gute Portierbarkeit und hohe Effizienz. Hier wird ein Maschinenmodell definiert und Bytecode für diese Virtuelle Maschine übersetzt. Der Bytecode kann interpretiert, binär in die Zielarchitektur übersetzt oder Just-in-Time übersetzt werden. Die Java Virtual Machine ist dabei die am weitesten verbreitete Bytecode-basierte Virtualisierung.

Im Bereich der Embedded Systems, insbesondere Industrial IoT und Automotive, spielen Virtualisierungslösungen zunehmend eine Rolle. Hypervisoren adressieren dabei die Bedürfnisse dieser spezialisierten, ressourcenbeschränkten Systeme. Diese haben jedoch Nachteile in der Portierbarkeit, da sie sehr spezifisch zugeschnitten sind und zwischen Geräten nicht immer direkt übertragbar sind.

Bytecode-basierte Virtuelle Maschinen, bei denen Anwendungen in Modulen gekapselt auf beliebigen Plattformen ausgeführt werden können, haben hier den Vorteil einer Separierung zwischen der Portierung einer Laufzeitumgebung und der Anwendung. So lassen sich Applikationen ohne weitere Anpassung unmittelbar auf unterstützten Geräten ausführen.

In den letzten Jahren ist dabei WebAssembly, das ursprünglich für die schnel-

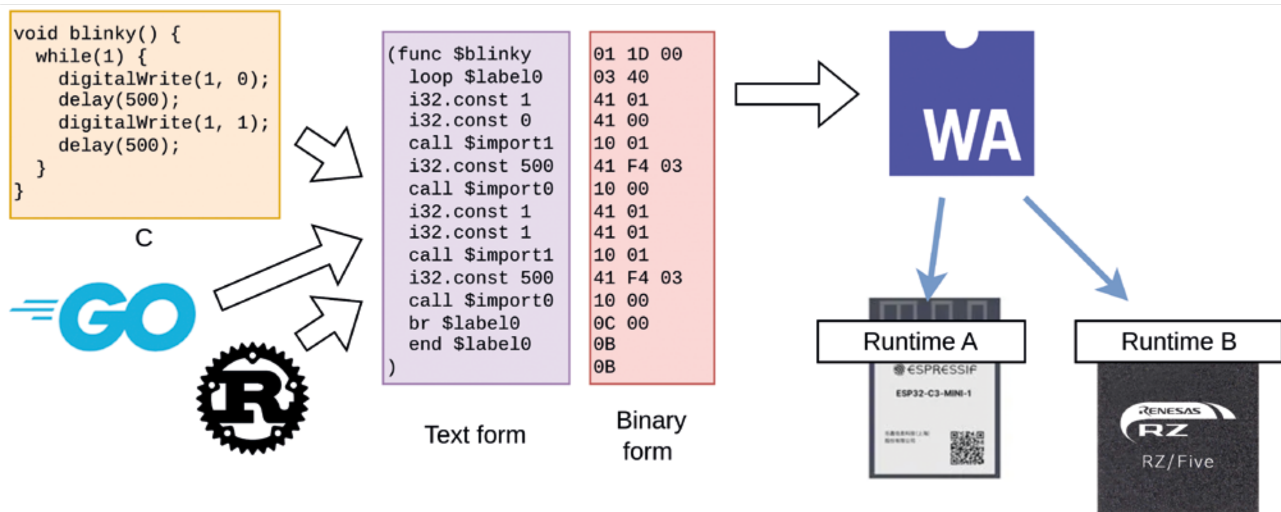

Bild 1: WebAssembly Übersetzung und Ausführung.

lere Ausführung von Applikationen in Browsern entwickelt wurde, immer stärker in den Fokus von Anwendungen jenseits des Browsers gerückt. Im Gegensatz zur Java Virtual Machine ist die virtuelle Maschine von WebAssembly maschinennäher und unterstützt sehr viele verschiedene Programmiersprachen als Ausgangspunkt. Mit diesen Eigenschaften ist es auch interessant für Embedded Systems.

Im Folgenden werden die technischen Grundlagen von WebAssembly kurz präsentiert und der Einsatz von WebAssembly in industriellen IoT-Anwendungen diskutiert.

Grundlagen von WebAssembly

WebAssembly wurde im Jahr 2014 als ein gemeinsamer Standard der großen Browser-Hersteller vorgestellt [1]. Die Motivation dafür bestand unter anderem in der zunehmenden Komplexität von Web-Anwendungen, die durch JavaScript nicht effizient adressiert werden, konnte. Insbesondere ist das bei rechenaufwendigen Aufgaben, wie zum Beispiel im Bereich Computer Vision oder der Echtzeitvisualisierung von Daten der Fall. Während es zu diesem Zeitpunkt bereits Ansätze wie asm.js gab [2], war hier das Ziel von vornherein ein portables, standardisiertes Format zu unterstützen, mit dem sich Applikationen im Client-Browser ausführen lassen. Dabei sollte weiterhin eine Interaktion mit dem Document Object Model (DOM) sowie JavaScript möglich sein. Gegenüber dem Ansatz, JavaScript zur Beschleunigung Just-in-Time zu übersetzen, hat WebAssembly den Vorteil, dass beliebige Programmiersprachen nach WebAssembly übersetzt werden können. Als Beispiel sei der Game-Boy-Emulator WasmBoy [3] genannt, der es erlaubt, im Browser Game-Boy-Spiele zu spielen.

WebAssembly wird als Standard des W3C-Konsortiums gepflegt und weiterentwickelt [4]. Basierend darauf ist die Bytecode Alliance als Non-Profit-Organisation entstanden, die Software für die Ausführung außerhalb des Browsers pflegt und Standards jenseits des Webs definiert [5]. Daneben ist ein extrem reichhaltiges Open-Source-Ökosystem entstanden.

Technisch handelt es sich bei WebAssembly um eine Stack-basierte virtuelle Maschine. Bytecode operiert auf diesem Stack, Operanden zu Operationen sind wie in Java immer oben auf dem Stack. Im Gegensatz zu Java gibt es (aktuell) keine Objekte und Garbage Collection, sondern ein flaches Speichermodell wie in C oder Rust.

Die Übersetzung und Ausführung von WebAssembly sind exemplarisch in Bild 1 dargestellt. Ausgangspunkt ist in diesem Beispiel ein einfaches Embedded Programm in C. Ebenso kann eine große Reihe weiterer Programmiersprachen genutzt werden. Dies ist besonders interessant, weil es Firmen mit großen Embedded-System-Projekten erlaubt, neue Programmiersprachen wie Rust einzuführen, auch wenn die Plattform selbst von keinem Rust-Compiler unterstützt wird.

Aus der Quell-Programmiersprache wird ein Bytecode in binärer Form (wasm) erstellt. Neben der binären Form existiert auch eine Textdarstellung (wat), die in Bild 1 dem binären Format gegenübergestellt ist. Ohne auf die genauen Kommandos einzugehen, ist das Format einer Stack-basierten virtuellen Maschine hier klar erkennbar.

Ein WebAssembly-Modul in binärem Format kann in einer beliebigen WebAssembly Laufzeitumgebung ausgeführt werden. Verbreitet sind dabei i) Interpreter, ii) Interpreter mit Just-in-Time Compilern, und iii) Ahead-of-Time Laufzeitumgebungen. Aufgrund der Ressourcenbeschränkungen von Embedded Systems fallen

CC BY

https://doi.org/10.30844/IM_23-3_48-51

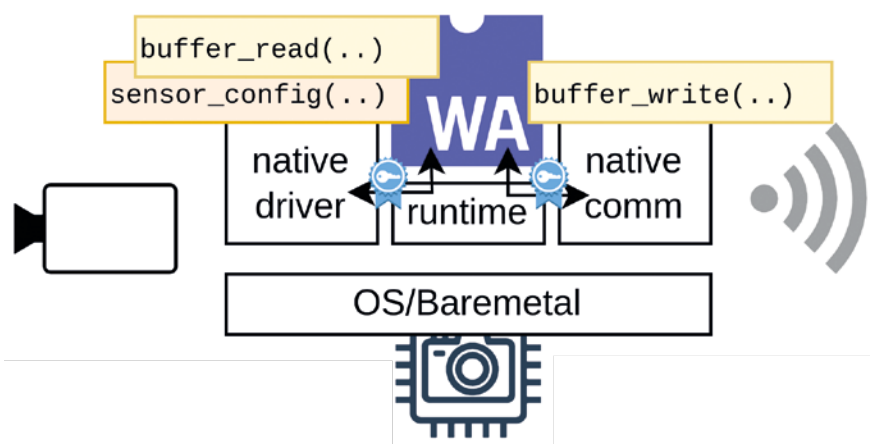

Bild 2: Schnittstellen zur Vernetzung von WebAssembly-Modulen.

Just-in-Time-Ansätze weg, sodass Embedded-fokussierte Laufzeitumgebungen meist Interpreter sind, die gelegentlich auch übersetzte Module erlauben. Ein Interpreter durchläuft den Bytecode dabei als Daten und führt für jeden Befehl eine Funktion aus. Wenn der Bytecode vorab (Ahead-of-Time) übersetzt wird, werden hingegen direkt Maschinenprogramme ausgeführt. Die gängigsten Interpreter für Embedded Systems sind aktuell die WebAssembly Micro Runtime der Bytecode Alliance [6] und WASM3 [7]. Diese Laufzeitumgebungen können entweder direkt auf der Hardware instanziiert oder in Betriebssysteme integriert werden.

Sichere Vernetzung von WebAssembly-Modulen

Ausgehend von den einzelnen Modulen ist interessant, wie man größere IoT-Systeme aufbauen kann. Ausgangspunkt ist dabei eine einzelne Laufzeitumgebung. Module können über ein generisches Interface-Format namens Wasm Interface Type (WIT) [8] untereinander oder mit der Außenwelt kommunizieren. Ähnlich wie POSIX definiert das WebAssembly-System Interface (WASI) [9] grundlegende Systemfunktionen. Weitere Interfaces werden über die Zeit spezifiziert.

WebAssembly-Schnittstellen verfolgen das Konzept der berechtigungsbasierten Sicherheit (capability-based security). Neben statischen Berechtigungen, die zum Start eines Moduls importiert werden können, spielen insbesondere dynamische Berechtigungen eine zunehmend wichtige Rolle. Diese Berechtigungen werden dann zur Laufzeit von einem Modul benötigt, um bestimmte Operationen durchzuführen. Nicht-veränderbare Referenzen erhöhen dabei die Sicherheit.

Beispielhaft für Anwendungen im Umfeld von Industrial IoT ist dabei in Bild 2 dargestellt, wie ein WebAssembly-Modul mit nativen Funktionen interagiert. Wir erwarten in den nächsten Jahren standardisierte Schnittstellen für die Interaktion von WebAssembly-Modulen, die auf Berechtigungen basieren. Im Bild sind Schnittstellen für die Sensor-Konfiguration (hier: Kamera) und die Kommunikation dargestellt. Solche Kommunikationsschnittstellen werden aufgrund der angestrebten Portabilität sicher abstrahiert und transparent über darunterliegende physikalische Kommunikation über Wifi, MQTT etc. durchgeführt.

Die Sicherheit der Kommunikation von WebAssembly-Modulen untereinander und mit der Außenwelt ist besonders wichtig. Hier arbeiten wir aktuell an sicheren Schnittstellen auf Basis von authentifizierten Kommunikationskanälen mit SPIFFE [10]. SPIFFE ist ein Protokoll, dass es Containern, also in unserem Fall Webassembly-Modulen, erlaubt, miteinander sicher zu kommunizieren. Das umfasst sowohl Authentifizierung als auch Transportsicherheit. Das Protokoll wird aktuell im Docker-Umfeld eingesetzt, wir arbeiten jedoch an einer Portierung für WebAssembly, die sich dann mit den Docker-Containern integriert. Gegenüber anderen Standards, bietet SPIFFE eine Integration über verschiedene Container-Formate hinweg.

Programmierung von Skalierbaren WebAssembly-Anwendungen

Wie einleitend motiviert, ist eines der Ziele für moderne IoT-Systeme, komplexe Applikationen auf eine heterogene Gerätelandschaft abzubilden, und dabei dynamisch in Bezug auf die Performance-Ansprüche zu reagieren. Darüber hinaus ist eine Resilienz gegen Ausfälle dringend notwendig. Mit den bisher beschriebenen Elementen ergibt sich eine solide technische Grundlage, auf der Frameworks für die Entwicklung, Verteilung und Überwachung von IoT-System aufbauen können. Ausgangspunkt ist heutzutage häufig die modellbasierte Software-Entwicklung, die es erlaubt, aus Modellierungssprachen Quellcode zu generieren.

Für den Einsatz im industriellen Umfeld könnte zum Beispiel ein Framework wie Simulink für Kontrollaufgaben oder eine Stream Processing Software wie KNIME [11] für Data-Science-Anwendungen genutzt werden. Durch die Generierung von WebAssembly-Modulen aus diesen Frameworks lassen sich damit portierbare und interoperable Applikationen erstellen, die sich leicht auf große IoT-Systeme abbilden lassen.

In Bild 3 ist beispielhaft ein Datenflussmodell dargestellt, welches in einer nicht näher spezifizierten Modellierungssprache entworfen wurde. Jeder der Knoten kann in ein WebAssembly-

Literatur

[1] Haas, A. u. a.: Bringing the web up to speed with WebAssembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (2017), S. 185-200.
[2] Herman, D. u. a.: asm.js. URL: https://asmjs.org, Abrufdatum 10.3.2023.
[3] Turner, A.: WasmBoy. URL: https://wasmboy.app/, Abrufdatum 10.3.2023.
[4] Schuff, D. u. a.: WebAssembly Working Group. URL: www.w3.org/wasm/, Abrufdatum 10.3.2023.
[5] Bytecode Alliance: About the Bytecode Alliance. URL: https://bytecodealliance.org/, Abrufdatum 10.3.2023.
[6] Bytecode Alliance: WebAssembly Micro Runtime. URL: https://github.com/bytecodealliance/wasm-micro-runtime, Abrufdatum 10.3.2023.
[7] Volodymyr Shymanskyy: WASM3. URL: https://github.com/wasm3/wasm3, Abrufdatum 10.3.2023.

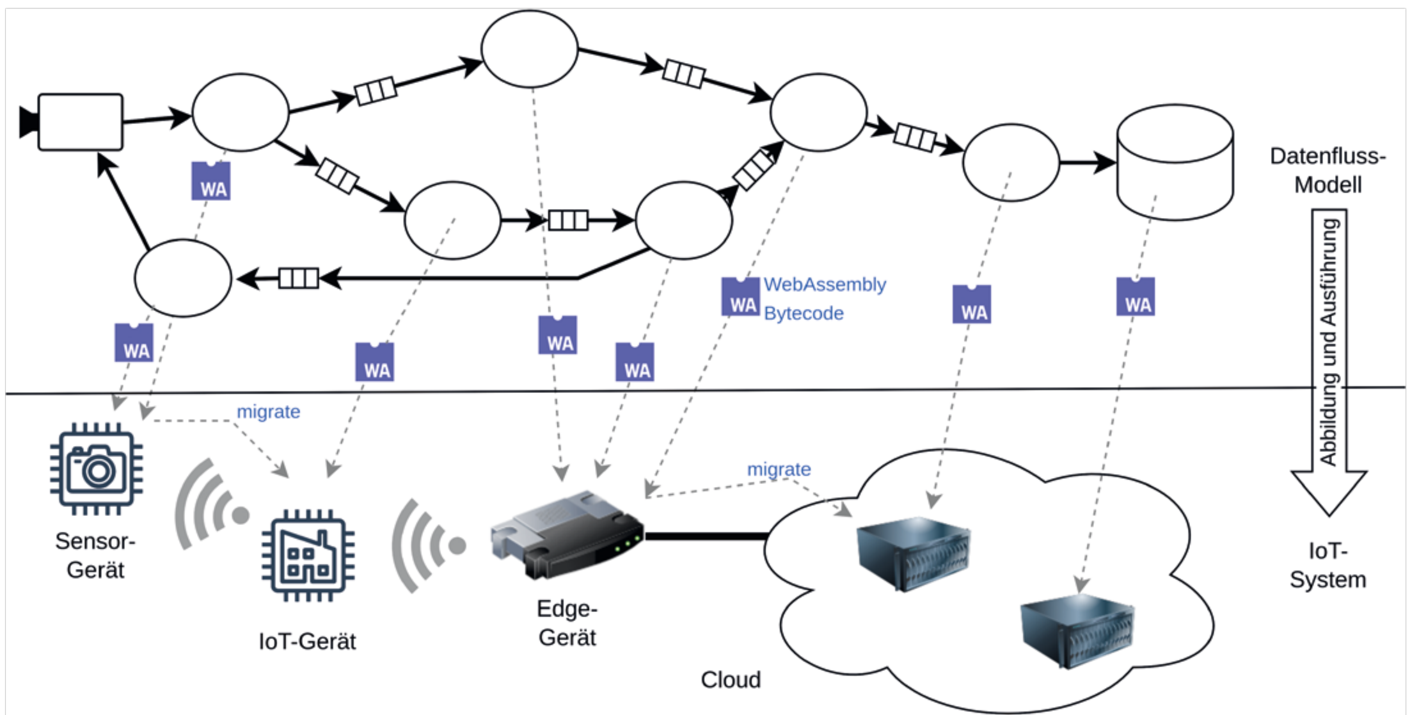

Bild 3: Vision des Mappings von WebAssembly Stream Processing auf eine IoT-Flotte.

Modul übersetzt werden und auf IoT-Geräte, Edge-Geräte und Cloud-Server verteilt werden. Die Kommunikationskanäle müssen auf die darunterliegenden Systeme und Standards abgebildet werden, sodass eine transparente Umsetzung ermöglicht werden kann. Die Verfügbarkeit sicherer Kommunikationsinfrastruktur, zum Beispiel basierend auf SPIFFE, ist dabei essenziell. Für das einfache Debugging könnten einzelne WebAssembly-Module auch vorübergehend im Browser instanziiert werden. Performance-Tuning kann aufgrund der Auslastung der Geräte erfolgen. Auf etwaige Flaschenhälse oder Ausfälle kann durch Migration der WebAssembly-Module reagiert werden. Durch die bestehende Hardwareabstraktion ist es so möglich, flexibel auch große Systeme zu entwickeln.

Eine Herausforderung der nächsten Jahre wird sein, dass die Performance von interpretiertem WebAssembly Bytecode leicht um den Faktor 10 schlechter sein kann als nativer, optimierter Code. Dieses Problem ist im Umfeld leistungsfähiger Plattformen mit verhältnismäßig viel Speicher (z. B. Browser oder Server) weniger signifikant, da Methoden wie Just-in-Time-Übersetzung in Maschinencode dies weitestgehend abmildern können. Die alternative Ahead-of-Time-Übersetzung scheint da besser geeignet für den Einsatz in Embedded-Geräten, erfordert aber tiefergehende Schutzmechanismen bei der Ausführung von Benutzercode, wie zum Beispiel über Code-Signierung. Gerade in überschaubaren Umgebungen kommt man damit auf bedeutend weniger Einfluss auf die Performance in der Größenordnung von ca. 30 % [12].

Fazit und Ausblick

WebAssembly ist eine äußerst interessante Virtualisierungslösung. Im Vergleich zu anderen Formen der Virtualisierung, die aktuell in Embedded Systems Verbreitung finden, ist es bedeutend portierbarer und im Vergleich zu anderen Bytecode Virtuellen Maschinen wie Java näher an den Speichermodellen von C und der darunterliegenden Hardware. Durch das reichhaltige Open-Source-Ökosystem sehen wir viele Potenziale – insbesondere auch als Ausführungsformat für die modellbasierte Software-Entwicklung.

Die Entwicklung von WebAssembly ist aktuell noch im Fluss, wie die Verwendung von wiederverwendbaren Komponenten und Objekte im Bytecode. Einige konzeptuelle Veränderungen scheinen dabei äußerst hilfreich für die Adaption in Embedded Systems. Auch sind standardisierte Schnittstellen noch nicht in der Breite vorhanden – ein großes Potenzial, sich zu engagieren. Interessant wird dabei insbesondere sein, wie standardisierte Schnittstellen für den Zugriff auf native Funktionen von Spezialhardware (z. B. GPUs) geschaffen werden können. Diese wären sicher Schlüsselkomponenten für die effiziente Ausführung von rechenintensiven Algorithmen (z. B. aus der Computer Vision) in WebAssembly-basierten Implementierungen. Vorbilder gibt es hier bereits mit WebGPU [13] und WebNN [14].

Schlüsselwörter:
Virtualisierung, WebAssembly, OT-Sicherheit, IoT

[8] Wagner, L. u. a.: The wit format. URL: https://github.com/WebAssembly/component-model/blob/main/design/mvp/WIT.md, Abrufdatum 10.3.2023.
[9] Hickey, P. u. a.: WebAssembly System Interface. URL: https://github.com/WebAssembly/WASI, Abrufdatum 10.3.2023.
[10] Cloud Native Computing Foundation. URL: https://spiffe.io/, Abrufdatum 10.3.2023.
[11] KNIME AG. URL: www.knime.com/, Abrufdatum 10.3.2023.
[12] Wallentowitz, S. u. a.: Potential of WebAssembly for Embedded Systems. 11th Mediterranean Conference on Embedded Computing (2022), S. 1-4.
[13] Ninomiya, K. u. a.: WebGPU. URL: www.w3.org/TR/webgpu/, Abrufdatum 10.3.2023.
[14] Hu, N. u. a.: WebNN. URL: www.w3.org/TR/webnn/, Abrufdatum 10.3.2023.

CC BY
https://doi.org/10.30844/IM_23-3_48-51