WOOD

Extending a WebAssembly VM with Out-of-Place Debugging for IoT applications

Carlos Javier Rojas Castillo
carlos.javier.rojas.castillo@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Matteo Marra
mmarra@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Jim Bauwens
jim.bauwens@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Elisa Gonzalez Boix
egonzale@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Abstract

Internet of Things (IoT) enables collaboration between humans and a diverse range of machines, including embedded devices and sensors. Software development of IoT applications is challenging given the distributed nature of the applications and the limited resources of some devices. This paper focuses on an extension to the WARDuino IoT platform that enhances debugging support, an integral part of the software development cycle.

Popular offline debugging techniques such as logs, dumps, or record & replay are not suitable for IoT devices as they impose too much overhead on devices and often miss contextual information on the root cause of bugs. Online debuggers seem more suitable for IoT since they enable developers to remotely debug devices, but suffer from the probe-effect, non-reproducibility issues and high latency.

In this paper, we explore an online debugging approach that deals with the constraints of IoT devices and enables low latency remote debugging. To this end, we bring ideas of out-of-place debugging, in which the state of a running application is moved to the developer's machine, to IoT. We implement our out-of-place debugging approach for IoT in *WOOD*, an extension to the WARDuino VM that executes Web Assembly on embedded devices. The paper focuses on WOOD's features including capturing, moving and reconstructing debugging sessions, as well as support for accessing remote resources and live code updating.

VMIL '21, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Keywords: datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
Carlos Javier Rojas Castillo, Matteo Marra, Jim Bauwens, and Elisa Gonzalez Boix. 2021. WOOD: Extending a WebAssembly VM with Out-of-Place Debugging for IoT applications. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 Introduction

In the past, the Internet was mainly composed of *fat devices* i.e. end-user devices with great computational power and memory capacity. Nowadays, the Internet is a heterogeneous environment home to different kinds of devices, ranging from *sensors* and *actuators* to *cloud servers* and *clusters*. The part of the Internet that focuses on *small devices* i.e. devices with restricted computational power and memory capacities like micro-controllers or sensors, is known as the *Internet of Things* (IoT) [13].

Since the first mentioning of IoT, the idea of interconnecting small devices with the internet has gained a lot of popularity due to its applicability across diverse domains (industrial, healthcare, environmental monitoring, and more). Because of this, IoT has become a cross-cutting concern at the core of many academic and industrial research [2]. As more and more developers target such systems, it is essential that the entire software development process for IoT applications is properly supported.

Debugging is an integral part of the software development process, but it is an area that is still challenging for IoT applications. A recent study[8] surveying 194 IoT developers reported that 74% of participants rely on access to the devices to test and debug the IoT application. Moreover, 62% of the survey participants agreed that it is challenging handling failures in a way that data does not get lost and the system

remains available. The main problem is that there is a lack of tools that can deal with the device constraints and distributed nature of IoT systems.

Debugging approaches can be classified in two main families: online, in which the debugger guides the execution of the application, or offline (or post-mortem) in which debugging happens after a failed execution. The simplest form of offline debugging is *log-based*, which require developers to add print statements to source code. Log-based debugging is typically undesirable as it may introduce errors into an application and affect the timing of the application [15]. Existing logging solutions for IoT are said to be inefficient to properly monitor all the device in the system [8]. Another popular offline technique is the use of *dumps* which give information at the point of the failure. The success of the dumps solely depends on a large number of contextual information recorded [15]; for IoT devices such large amounts are voluminous given their restricted memory sizes. Finally, *record* and *replay* debugging enables a determinstic replay of past executions (that may contain the bug). However, they usually cannot cope with non-deterministic input coming from external resources (e.g. sensors) [12].

In this paper, we argue that online debuggers are more suitable for IoT environments since they can deal with external resources and typically impose less memory overhead than their offline counterparts. In particular, *remote online debuggers* allows for debugging IoT devices remotely, without requiring physical access to the devices. However, online debugging comes at the cost of higher latency and thus increasing the chance of a problem akin to Heisenberg effect called the *probe effect*. *Out-of-place debugging* [10] is a promising approach which supports remote debugging with low latency. It does so by transferring the remote execution state to a process running at the developer's machine where the debugging session is created. Debugging happens locally with an online tool as the one commonly found in IDEs for languages like Scala or Python Developers can then synchronize the code updates back to the remote device once the bug is fixed at the end of the debugging session.

In this paper we explore an out-of-place debugging approach for IoT environments. Our approach entails less network overhead since while being debugged the remote devices are not being touched, yet it allows developers to access remote resources (.e.g. sensors) when necessary. We focus on the different modifications we applied to WARDuino [6], a Web Assembly virtual machine for embedded devices to enable our debugging model. We show how we extend the built-in WARDuino debugger to provide all the information necessary that is needed to create a copy of the state of the device, and how this information is then used by a local process on the developers machine to replicate the remote state in the form of a debug session. Finally we explain how remote resources are handled by means of proxies and in what way they are significant to the debugging of IoT applications.

2 Background & Motivation

In this section, we describe the necessary background on which we base our work. In particular, we first describe WARDuino, the Web Assembly VM that we extended for this paper, and then the main concepts of out-of-place debugging.

2.1 WARDuino

WARDuino is a stack-based VM that enables programming for micro-controllers in Web Assembly. Similarly to other runtime environments for micro-controllers such as MicroPython and Espruino [4, 5, 14], WARDuino is uploaded in the micro-controller where it executes applications. Applications can be written in a wide range of languages (e.g. Rust, C/C++, JavaScript, etc.) and compiled into Web Assembly (or more specifically Wasm) by an external compiler. We can then install WARDuino into a micro-controller to execute the resulting Wasm.

In contrast to a high level VM such as the Java Virtual Machine (JVM), WARDuino has several features that makes it more suitable for resource constrained devices as the found in IoT:

- WARDuino manages to limit its memory impact on the micro-controller (i.e. the space it takes in the micro-controller), by letting developers configure the used functionality [6]. This prevents the need to upload unused functionality and thus reduces memory impact; which contrasts with the *all or nothing* approach of the above-mentioned runtime environments.
- Wasm is designed with performance in mind, which is positively reflected in WARDuino. Micro benchmarks demonstrate that WARDuino executes faster compared to existing runtime environments for micro-controllers [6, 7]. For example, five times faster than Espruino.
- By leveraging on Wasm and WARDuino, a wide range of high-level programming can be executed and managed using the remote debug and live code update facilities [6].

To execute the application on an embedded device, we need to flash the WARDuino VM along with a copy of the compiled Wasm bytecode to the device. WARDuino then identifies the main portion of the application and executes it. Currently, to interact with the WARDuino VM, interrupt commands can be sent to the device over a serial connection that can pause the computation and control debugging aspects of the virtual machine.

2.2 Out-of-Place Debugging

In prior work, some of the authors have proposed out-of-place debugging for Big Data applications [11]. Out-of-place debugging is an online remote debugging model that envisions two main components: the *Debugger Monitor*, which is

executed alongside the application and the *Debugger Manager*, which runs in an external process and that provides the developer with a UI for debugging (typically executed at the developer's machine). When a breakpoint hits or an exception occurs in the application, the Debugger Monitor extracts the execution context of the program and generates a debugging event that is sent to the external Debugger Manager. The Debugger Manager then creates an online debugging session on the extracted execution context at the external process. The concept of out-of-place debugging also includes live code updating of the debugged remote application to be able to *patch and continue* the execution.

An out-of-place debugger aims to mitigate network latency issues by making debugging a local activity. Moreover, this embraces the concept of debugging errors in isolation. In particular, performing *local* online debugging on the remote application is possible by;

1. assuming that the target application also runs locally i.e. a copy of the target application runs in the debugger process.
2. extracting a *debug session* (i.e. the program and application state) from a running target application. And provide it to the debugger process to use it as if it were a locally retrieved session. In doing so it becomes possible to synchronise the execution of the local application, to the execution of the target application. As such, debugging becomes an in-place activity with reduced latency since any debug action is performed upon the local application and no longer requires network communication.

The only implementation of an out-of-place debugger is IDRA for Pharo Smalltalk [9]. The language environment plays a key role in the implementation of IDRA since Smalltalk is equipped with several features that ease the realisation of the architecture. For example, the construction of a debug session relies on the reification of the call stack by means of *thisContext*. Additionally, local debugging in IDRA happens through the Pharo Debugger.

2.3 From Out-of-Place to Out-of-Things

Out-of-place debugging exhibits great potential for an IoT context due to its ability to allow local debugging of remote processes. From an IoT perspective this is very interesting since local debugging ensures that resource-constrained devices are not overloaded with debugging activities (e.g. generating events for a trace or storing logs). Out-of-place additionally introduces conceptual mechanisms to access non-transferable resources and commit changes that become permanent on the remote processors, which are also practical in the IoT domain, as devices have resources (e.g. sensors) whose access would be beneficial during local debugging.

As is, however, out-of-place debugging is not directly applicable for IoT. For example, non-transferable resources are accessed transparently, which may impose too much of a network overhead and performance impact on resource restricted IoT devices. Therefore we introduce out-of-things debugging that extends and modifies out-of-place debugging to fit the resource constrained environment of IoT.

In this work we implement an out-of-things debugger on an existing VM for embedded devices, WARDuino. WARDuino is a suitable runtime to build on our prototype since it features (1) online debugging functionality that we can extend for out-of-things debugging and (2) can be compiled to run on the developer's machine besides IoT hardware, allowing us to execute the VM locally and thus potentially support local debugging of applications.

We call *WOOD* (WARDuino out-of-place Debugging) the VM that embodies all the changes on WARDuino for enabling out-of-things debugging, and we call *local WOOD* the local version running on the developer's machine. In what follows, we will elaborate on the concrete design of WOOD.

3 WOOD: Enabling Out-of-Things Debugging

In this section we first describe the general architecture of out-of-things debugging, and then discuss our approach to deal with the debugging session, accessing remote resources, and dynamic module and state update.

3.1 Out-of-Things Debugging Architecture

Figure 1 shows the main components of an out-of-things debugger. Similar to out-of-place debugging, out-of-things debugging involves two processes, the application process (i.e. WOOD) and the debugger process, to support debugging a remote process. Additionally, the out-of-things architecture features one additional process called *local application process* (i.e. local WOOD) that represents the local execution of the VM, in other words, the VM needed to locally simulate and debug the remote device. This process becomes responsible for the roles of *Local Debugger Monitor* and *Local Updater*, needed to debug and update the local execution.

In the setup shown in Figure 1, the debugger and local application processes run on the developer's machine, whereas the application process runs on the target device. The communication between the processes on the developer's machine and the target device happens over the network, whereas the communication between the debugger and the local application process happens through an interprocess mechanism.

While an application is running on the application process (1), the *Debugger Monitor* monitors its execution. When a breakpoint is reached or an exception is thrown, the Debugger Manager then constructs a debug session (2), serializes it, and sends it across the network (3) to the *Debugger Manager* at the developer's machine.

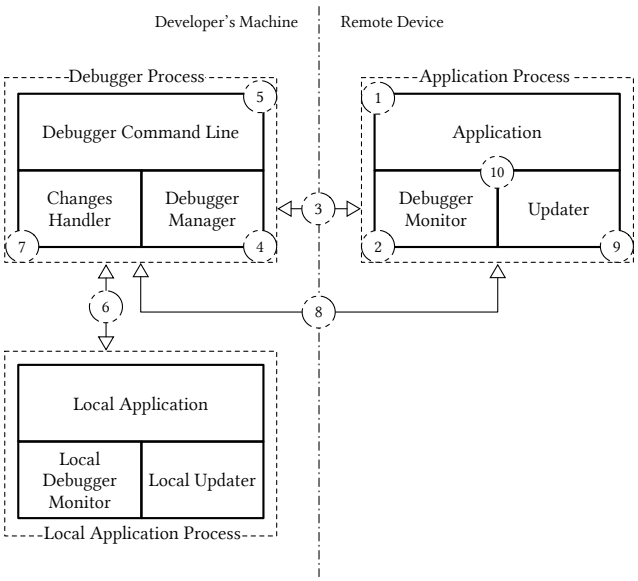

Figure 1. The architecture of the out-of-things debugger; dashed square indicates a process; the numbers indicate the order of activities after a breakpoint is reached or exception occurs; arrow line indicates process communication.

When the debug session is received (4), the *Debugger Manager* deserializes it and forwards it to the debugger client (5). The Debugger Manager also sends a copy to the local application process (6) which is used to synchronise the application and execution state of the local application process to the application process. Deserializing the received debug session boils down to creating a debug session at the developer's machine with the same runtime conditions to those of the target application. From this moment on local online debugging can occur, i.e debug operations are continuously forwarded to the local application process which in turn get processed appropriately. For example, adding a breakpoint, stepping through the execution, is handled by the local Debugger Monitor and applied on the local application.

Once the developer is ready to commit a bug fix (7), the *Changes Handler* transmits source code changes made to the Updater component (8). And in doing so it also ensures that the changes do not break type safety of the program [1].

The *Updater* component is responsible to apply any change send by the Changes Handler to the target application (9). Once a change is applied, the target application resumes the program execution (10). For instance, if at some point during debugging the developer has decided to commit the changes in the debugger process, the Updater will replace the running application with a newly provided one.

[1]Web Assembly is a type-safe language and thus changes to the Wasm source code might break the type safety.

On the whole, the *Updater* allows to dynamically apply changes at different levels of granularity, for example, without replacing the existing application, one can also change the execution state of the application.

3.2 The Debugging Session

Capturing a debug session is one of the most important operations when it comes to implementing out-of-things. To support this we extended WARDuino with the ability to extract additional state information during debugging.

In the following we enumerate all the information that composes a WOOD debug session; the asterisk indicates what is (partially) reused from WARDuino:

(*1) the *program counter*
(*2) *call stack*: the different functions called (i.e. trace) to the point where the debug session is created. Additional data pointers were added to the data structure to keep track of the blocks related to the call frame.
(*3) all the *breakpoints*.
(4) the *error counter*: keeps track of a program location where an exception previously occurred. More specifically, when an exception occurs on a target device, WOOD restarts the execution of the program. The error counter keeps track of which instruction caused the program to fail.
(5) *values stack*: the values on the stack used throughout the execution of a program (e.g. arguments to function calls, a place where local variables are stored, etc.).
(6) all the *global* values
(7) all the information related to Wasm *tables* i.e. initial and maximum size of the table, as-well-as, the table entries. Tables is the way how Wasm supports function pointers.
(8) a copy of the *memory* pages; a memory page is the way how Wasm provides heap memory space to an application.

As shown above, WOOD's debug session is larger than WARDuino's since it incorporates all application and execution state of a device. This is of course expected as WOOD's debug session enables local debugging of remote applications, in contrast to WARDuino's debug session which only allows for remote online debugging.

Reconstructing the Execution. WOOD provides the infrastructure for local debugging of a remote exception, a key feature of out-of-things debugging. It does so by transferring a debug session between the local WOOD and the remote one. In practice, WOOD is able to replace its current application and execution state with the one provided by a debug session, thus enabling local debugging.

Compared to WARDuino, this feature is completely new and brings along several debugging conveniences. For example, one can easily switch from one debug session to another i.e., enables *debug session versioning*. It also makes *patching*

at the level of application or execution state possible. As explained later on in section 3.4, a developer can (for example) change stack values in a debug session, which eventually can be committed to the remote device or local application process.

3.3 Accessing Remote Resources

In the context of out-of-place debugging, Marra et al. [10] employ *proxy objects* as a way to access non-transferable-resources. In our work, examples of non-transferable-resources are sensors or other hardware components commonly only present on IoT devices. Since those resources are only accessible by means of function calls, we designed WOOD with the ability to perform proxied function calls. We also use proxies as a way to access such resources but instead of proxying all the calls to an object by default, we introduce a remote function invocation mechanism known as *proxy call*, that when performed gives access to specified resources of a target device. This way, when local WOOD is about to invoke a function locally that is marked as a remote resource, a remote invocation happens instead.

The general idea of proxy calls is that when performed, the call traverses the network and asks a target device to execute a function. The request might also contain arguments needed by the remote function. Once the execution completes, the resulting value of the call is then returned to the caller.

However, to correctly enable proxy calls, we need to keep the constrained characteristics of devices in an IoT environment in mind:

- Despite featuring different network technologies, communication is challenging for those devices (e.g. limited communication range, harmful for the device's lifespan). And thus performing proxy calls is not always possible or should be minimized.
- Proxy calls should have almost no effect on the target application. Unfortunately, no effect is not possible since proxy calls may result in side effects. Nevertheless, we still need to isolate the execution of proxy calls from the execution of the target application. This is especially useful when dealing with exceptions. For example, when debugging a production application, a proxy call can raise an exception that compromises the running application.

Request & Answers Proxy Calls. Taking those characteristics into account, when performing a proxy call, local WOOD sends the arguments and the identity of the function to call to WOOD. When receiving the proxy request, WOOD temporarily pauses the execution of the current program, then executes the function using the received arguments and returns the result value of the call (if any). If the call results in an exception, the exception message is answered instead.

Note that WOOD is the only one able to answer proxy calls but not request them. This is because requesting proxy calls is only needed by local WOOD. And since the needed functionality (to support the request of proxy calls) takes memory space in the memory restricted IoT device, we simply omit it.

Configuring the Debugger to use Proxies. The out-of-things debugger uses a *debugger configuration file* to know which functions to proxy. For this, the developer adds a *proxy* entry in the file, providing an array of names for the functions that need proxying. This information is then forwarded to local WOOD and used to perform remote invocation whenever necessary.

3.4 Dynamic Module and State Update

In an out-of-things debugger, when the developer changes the source code and commits the change, the Changes Handler transfers a compiled version of the source code to the Updater (i.e. WOOD). To enable live code updating, WOOD can dynamically replace a Wasm module with another one, meaning that, if WOOD would be currently executing a Wasm application, it can stop the application's execution and replace it with another one.

This particular way of updating software is an example of *live code update*. The advantage of this approach is that there is no need to physically access the device in order to flash a new codebase in it. Additionally it is also faster when compared to flashing over a serial connection [17].

3.4.1 Modify Application and Execution State.

Besides changing the source code and committing these changes, developers can modify parts of the execution state during debugging. For example, a developer can choose to patch the extract the current state of local WOOD and patch it into WOOD.

What follows describes what can be changed:

- *Values stack*: the values stack contains all the values used during the program execution of a Web Assembly program. For example, it can contain the arguments used for a function call. Thanks to this debugging feature it becomes possible to change those arguments dynamically.
- *Globals*: developers can change the values stored within the global variables.
- *Table*: a table in Wasm is the way how it supports function pointers. Such a table contains entries that correspond with function identifiers. During program execution, one can dynamically specify an index in the table, that tells what function will be called (i.e. the function identifier). The out-of-things debugger lets the developer update entries in that table, thus replace a function identifier with another.

The ability to modify these data structures is useful when one wants to perform low-level modifications to the execution state. However, it may endanger the program's execution given that type safety can no longer be guaranteed: e.g. it is possible for developers to replace functions in the table with other functions using different type signatures, or to modify stack values from float to integer. To prevent such issues, the debugger currently only allows changes that preserve type safety (but this can be disabled by an experienced developer).

4 Open Challenges

We now describe some challenges that remain in order to improve the performance and practicality of WOOD.

Performance optimizations. Initial benchmarks show that the execution speed of WOOD is lower than that of the original WARDuino. This is mainly due to the current implementation of the network socket server. Right now the socket server is responsible for 1) listening for network activity and 2) processing the network activity if any occurs. Since communication to the device can happen at any time, the socket server is regularly polled by the interpreter loop of the VM. This significantly reduces performance of instruction execution. Fortunately, performance is expected to improve when decoupling these two processes. For this, we plan to register the server's responsibility as an interrupt handler, where the interrupt gets trigger when network activity occurs. Thus the network stack of the IoT device becomes responsible for listening for network activity and triggering interrupts. This would result in a significant speedup of WOOD in the ESP32 family of devices we used for testing as they feature a dual core architecture with one core responsible for network activity.

Mapping debugging operations with WASM source symbols. Source mapping enables developers to perform debugging operations at the level of the debugged source code. Right now we support source mapping on symbols from Wasm files. To realise source mapping, we used several external tools (i.e. *The Web Assembly Binary Toolkit* and a modified version of it) [1, 16], that generate relevant debug information (e.g., quantity of defined functions, global variables, etc.) while compiling the source code. Moreover, we parse this information and capture it as an instance of *WAModule class*.

However developers do not typically implement their applications in Web Assembly, but rather they compile to Web Assembly from other higher level languages (e.g. C++, Rust, AssemblyScript, etc.). As such, our debugger should be used directly from these higher level languages. Unfortunately, currently there exists no stable standardised mapping to Wasm. Work is underway to support DWARF debug info with Web Assembly [3], but this is not yet complete. In future work we will look at how these projects evolve and how they can be integrated with this work.

Proxy granularity. As explained in Section 3.3, our debugger allows for the proxying of function calls between the local WOOD process and a WOOD application running on a remote IoT device. Right now developers need to specify what function calls are proxied. This means that it is up to the developer to decide at what granularity a particular segment of code can be debugged, which can be problematic as developers can accidentally proxy too much calls (incurring in high network overhead) or too few calls (potentially missing bugs located in the proxied code). To assist developers in this decision, we are considering some forms of static analysis. Another idea is to allow mixed proxy calls depending on the stack trace. For example, a function call to *digitalWrite* (a function that can toggle the set of a GPIO hardware pin) will only be proxied if a particular function can be found in the stack trace.

5 Conclusion

In this paper, we explored an online debugging approach that can deal with the constraints of IoT devices and enables low latency remote debugging. To this end, we introduce out-of-things debugging, a variant of out-of-place debugging specially designed for IoT, in which the state of an application running in a device is moved to the developer's machine so that it can be debugged locally with the classic online debugging facilities offered by mainstream IDEs. By moving the process locally, all debugging operations have low latency and do not impact the remote IoT device, limiting downtime. *WOOD*, our out-of-things debugger, is implemented as an extension to the WARDuino VM that executes Web Assembly on embedded devices. While WOOD already runs on the ESP32 family of embedded devices and can be used to debug remote applications locally and out-of-place on the developer's machine, further research is needed to tackle several open challenges and make WOOD a practical solution to debug IoT applications.

References

[1] Web Assembly. [n.d.]. The WebAssembly Binary Toolkit. https://github.com/WebAssembly/wabt. Accessed: May 1, 2021.
[2] Luigi Atzori, Antonio Iera, and Giacomo Morabito. 2010. The Internet of Things: A Survey. *Computer Networks* 54, 15 (Oct. 2010), 2787–2805. https://doi.org/10.1016/j.comnet.2010.05.010
[3] Yury Delendik. [n.d.]. DWARF for WebAssembly. https://yurydelendik.github.io/webassembly-dwarf/. Accessed: 10 August 2021.
[4] Espruino. [n.d.]. Espruino. https://www.espruino.com/. Accessed: April 20, 2021.
[5] Damien George. [n.d.]. MicroPython. https://micropython.org/. Accessed: April 20, 2021.
[6] Robbert Gurdeep Singh and Christophe Scholliers. 2019. WARDuino: a dynamic WebAssembly virtual machine for programming microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International*

Conference on Managed Programming Languages and Runtimes - MPLR 2019. ACM Press, 27–36. https://doi.org/10.1145/3357390.3361029

[7] Minsu Kim, Hyuk-Jin Jeong, and Soo-Mook Moon. 2016. Small Footprint JavaScript Engine. In *Components and Services for IoT Platforms*. Springer International Publishing, 103–116. https://doi.org/10.1007/978-3-319-42304-3_6

[8] Amir Makhshari and Ali Mesbah. 2021. IoT Bugs and Development Challenges. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 460–472. https://doi.org/10.1109/ICSE43902.2021.00051

[9] Matteo Marra. [n.d.]. IDRA. https://gitlab.soft.vub.ac.be/Marra/IDRA. Accessed: May 1, 2021.

[10] Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. 2018. Out-Of-Place debugging: a debugging architecture to reduce debugging interference. *The Art, Science, and Engineering of Programming* 3, 2 (nov 2018). https://doi.org/10.22152/programming-journal.org/2019/3/3

[11] Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. 2020. A debugging approach for live Big Data applications. *Science of Computer Programming* 194 (2020), 102460. https://doi.org/10.1016/j.scico.2020.102460

[12] Charles E. McDowell and David P. Helmbold. 1989. Debugging Concurrent Programs. *Comput. Surveys* 21, 4 (Dec. 1989), 593–622. https://doi.org/10.1145/76894.76897

[13] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. 2012. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks* 10, 7 (2012), 1497–1516. https://doi.org/10.1016/j.adhoc.2012.02.016

[14] Duktape organization. [n.d.]. Duktape. https://duktape.org/. Accessed: may 1, 2021.

[15] David Pacheco. 2011. Postmortem Debugging in Dynamic Environments. *Commun. ACM* 54, 12 (Dec. 2011), 44–51. https://doi.org/10.1145/2043174.2043189

[16] TOPLLab. [n.d.]. The Webassembly Binary Toolkit. https://github.com/topllab/wabt. Accessed: May 1, 2021.

[17] Chi Zhang, Wonsun Ahn, Youtao Zhang, and Bruce R. Childers. 2016. Live code update for IoT devices in energy harvesting environments. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. 1–6. https://doi.org/10.1109/NVMSA.2016.7547182