A Framework for Web Application Integrity

Pedro Fortuna[1], Nuno Pereira[2] and Ismail Butun[3,4]
[1]*Jscrambler, Rua Alfredo Allen, 455, 4200-135 Porto, Portugal*
[2]*DEI/ISEP/IPP, Department of Computer Engineering, Polytechnic Institute of Porto, Porto, Portugal*
[3]*Department of Computer Engineering, Abdullah Gul University, Kayseri, Turkey*
[4]*Department of Information Systems and Technology (IST), Mid Sweden University, Sundsvall, Sweden*

Keywords: Web Application, Application Security, Obfuscation, Execution Integrity, Data Integrity.

Abstract: Due to their universal accessibility, interactivity and scaling ease, Web applications relying on client-side code execution are currently the most common form of delivering applications and it is likely that they will continue to enter into less common realms such as IoT-based applications. We reason that modern Web applications should be able to exhibit advanced security protection mechanisms and review the research literature that points to useful partial solutions. Then, we propose a framework to support such characteristics and the features needed to implement them, providing a roadmap for a comprehensive solution to support Web application integrity.

1 INTRODUCTION

Web applications had a significant evolution in the last decade. They went from applications with very little interactivity, where the user would explicitly submit each request to the server and wait for the response, to the exceptionally interactive applications of today, which rival with native applications. In large part, this evolution is due to a model based on the execution of Web application code on the browser. Besides the interactivity that executing code on the client's browser allows, it also enables Web applications to scale more easily.

Web applications that follow the browser execution model are perhaps the most common form of delivering software nowadays. A natural desire to have uniform application delivery and development is leading to the appearance of the Web application model based on client-side code execution in many different areas and devices. Web applications however still face serious security challenges, and client-side code execution presents particular difficulties that are not trivial to overcome. When the code executes on the client, measures implemented on the server to protect the application against certain types of attacks are of little or no use (Nava and Lindsay, 2009), as the data flow of attacks frequently does not involve the server. Threats to client-side code execution can arise from, for example, malicious browser extensions (Kapravelos et al., 2014), or third-party code included by the Web application (a common practice related to revenue models based on advertisement networks), which create trust relationships that attackers can exploit (Nikiforakis et al., 2012). Section 4 discusses these and other threats more systematically.

While client-side Web application protection is not a new research theme, we propose two important directions: (i) a comprehensive framework providing a level of protection that is not possible with partial solutions (which we will review in Section 5), and (ii) include in the framework features for self-protection, self-healing and data integrity. Another important feature is that our framework relies on protections being delivered with the application code. This facilitates the delivery of up-to-date protection without assuming a particular execution environment other than a standard Web application execution environment.

1.1 Organization

In the following section (Section 2), we will start by developing further why we think Web applications are entering new realms of IoT-based applications. This motivates the need for Web applications that provide advanced security guarantees (self-healing, self-protection and data integrity). Next, in Section 3,

Fortuna, P., Pereira, N. and Butun, I.
A Framework for Web Application Integrity.
DOI: 10.5220/0006720204870493
In *Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP 2018)*, pages 487-493
ISBN: 978-989-758-282-0
Copyright © 2018 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

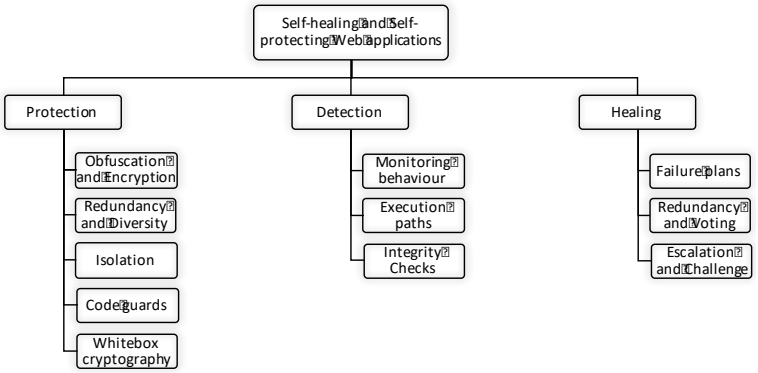

Figure 1: Techniques Towards Self-healing and Self-protecting Applications.

we will provide an overview of the research literature concerning self-healing, self-protection and data integrity. Section 4 will present the threats against Web applications with client-side code execution and formalize our threat model. Section 5 introduces our framework for Web Application Integrity and finally, Section 6 provides conclusions.

2 WEB APPLICATIONS EVERYWHERE

In this section, we will present several examples that support Web applications are going beyond traditional scenarios. Not only we can find today many devices and platforms that support this type of applications, but also there is an increasing trend to develop them for different areas of everyday life.

Traditionally, embedded systems were rather closed to the outside world, having very limited networking options. However, this is increasingly not the case. Most embedded systems today can be globally connected using the Internet Protocol (IP) standard, and take part of what became known as the Internet of Things (IoT). Developers of systems for the IoT soon started employing the same standards used to develop Web applications, giving birth to yet a new term: the Web of Things (WoT) (Atzori and Morabito, 2010). By developing WoT applications, developers tap on the already available protocols, libraries (e.g. HTTP, Websockets, JSON), the large amount of trained developers, and are also able to benefit from the architectural advantages of the Web application model, such as scalability and ease of update which is important problem of today's IoT (Schneier, B., 2014).

Javascript plays a central role in the development of WoT applications. While traditionally its adoption in embedded devices was dismissed due to its increased requirements on computing resources, we can observe that many platforms enabling developers to create WoT applications have appeared recently (e.g. ("The Tessel Board," 2017)).

A testament to the advantages of using existing protocols for the WoT is also the amount of Web application development tools for this type of devices that appeared very quickly. The Cyclon.js (Cylon.js, 2017) is a JavaScript framework for robotics and WoT applications, currently supporting 43 different platforms. IoT.js (IoT.js, 2017) is a framework for application development, based on JerryScript, a lightweight JavaScript engine, both open sourced by Samsung. Pi.js (Pi.js, 2017), is a cloud-based platform that supports writing JavaScript applications for the Raspberry Pi. These are just a few illustrating examples of the plethora of available tools for WoT application development. Naturally, along with these efforts, we can see the development of many emerging application domains from wearables, home automation, manufacturing, building management, and many other domains (Raggett, 2015).

3 RELATED WORK

We will now present an overview of techniques that can be relevant for developing self-healing and self-protecting Web applications. We have three main classes of techniques: protection, detection and healing, as depicted in Figure 1. This is not an extensive review of all existing techniques, but rather an overview of the more relevant techniques in the context of our work.

3.1 Protection

Code Obfuscation and Encryption: code obfuscation is the process of transforming an original source

code into a form that is much harder to understand and to debug, and assuring at the same time that the transformed code maintains the original functionality intact. While obfuscation, as a security approach, is generally not considered a strong and proven defence as encryption. However, recent work (Garg et al., 2013) established that obfuscation could theoretically be as secure as encryption in some applications, and this is an important development. Code obfuscation can also be combined with code encryption or code encoding techniques to make the resulting code more difficult to tamper with.

Redundancy and Diversity: pertains to techniques that use redundant program instances to force adversaries to manipulate more than one instance to be successful (Cox et al., 2006). Techniques that try to remove some of the predictability of the program execution and location of code and data as a barrier to program manipulation (Larsen et al., 2014). An important example is code polymorphism, which aims to defeat automated tampering attacks by frequently changing the aspect of the code.

Isolation: isolation is a fundamental technique in modern operating systems. This technique is useful, for example, to encapsulate application modules that need different sets of privileges, and are a way to delineate trust boundaries.

Code Guards: is a technique that consists in spreading multiple checks throughout the code, usually benefiting from code obfuscation. These checks enforce some restriction and may also defend themselves mutually (Chang et al., 2001).

White-box Cryptography: is a technique for protecting cryptographic code deployed to uncontrolled environments or devices (Chow et al., 2002). The protection is achieved by hiding the key using mathematical operations.

3.2 Detection

Some self-healing and self-protecting capabilities are triggered by the detection of some threat. In order to perform detection several techniques can be distinguished.

Monitoring Behaviour: the execution of the application can be constantly monitored by an external module (for example, the operating system or the browser in the case of Web applications), and this execution can be compared against a model of the normal execution of the application. This model can, for example, be a description of the expected interactions between system resources (Huang et al., 2008). There are also examples of building execution models using machine-learning techniques that enable the classification of malicious web applications (Borgolte et al., 2013).

Execution Paths: the execution path of an application can be an instance of monitoring behaviour by an external module, or the execution path can be controlled by inserting code checks into the application itself, to ensure that the code execution path is legitimate. One example of this technique is the Control Flow Integrity (Gekas et al., 2014) employed to protect native applications.

Integrity Checks: aim to detect if the application has been tampered, e.g. (Li et al., 2009). Integrity checks can be built into the code or done remotely, but in both cases, they usually rely on strong assumptions about the execution of the verification code, usually employing self-check-summing techniques. One important area is data integrity, where it is often assumed that secure communication channels can alone provide adequate guarantees. However, there are threats to web applications that can bypass secure channels (our threat model in Section 4 includes such scenarios), and several work approached this problem with both client-side and server-side solutions (Hallgren et al., 2013), (Karapanos et al., 2016).

3.3 Healing

In regard to the mechanisms to recover from attacks, we start by noting that many protection mechanisms are designed to cause the application to fail irrecoverably in order to stop an attack (e.g. (Oishi and Matsumoto, 2011)). While this approach is suiting for many applications, it is not an option for safety-critical applications.

Failure Plans: applications can be designed with handlers for expected security exceptions, and, instead to irrecoverable failure, these handlers may attempt to get the application back to a safe state. Theory in development and analysis of software safety plans is an extensive area, with many previous useful results (Ravikumar and Subramaniam, 2016).

Redundancy and Voting: Redundancy coupled with a mechanism to decide on the correct output (such as voting) is a well-known mechanism for healing and recovery (Latif-Shabgahi et al., 2004).

Escalation and Challenge: security escalation by triggering other defences or by presenting a challenge that only a legitimate user will be able to pass is also a widely used technique.

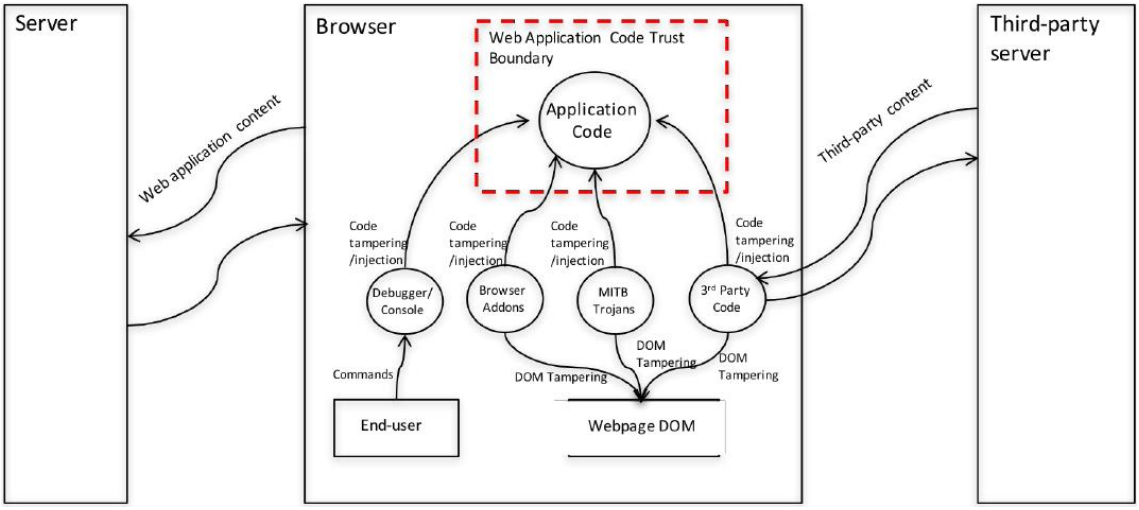

Figure 2: Threat Model and Trust Boundary.

4 THREAT MODEL

We assume the *hostile host model*, widely used in previous work on tamper-resistant software (Collberg and Thomborson, 2002). The attacker is capable of inspecting, tamper or inject code, to steal sensitive information that can be used in a broader scale attack. This is also an attack on the user's privacy (provided that the application manipulates or receives user data). The attacker can also apply deception techniques by changing the code to manipulate the messages that are presented to the user of the application.

Figure 2 depicts our threat model. The attacker can employ debugging tools, malicious browser extensions, Man-in-the-Browser Trojans (other than browser extensions, e.g. API hooking or malicious JavaScript), or can compromise third-party code included in the application (from, e.g. advertisement networks). These tools can allow the attacker to analyse, manipulate or inject code, and also manipulate the webpage content and its object-oriented representation – the Document Object Model (DOM). Our main goal is then to enforce the web application trust boundary protecting the web application execution from malicious manipulation on the browser platform.

5 WEB APPLICATION INTEGRITY

In order to enforce the Web Application trust boundary depicted in Figure 2, we need to protect the application execution from other code and plugins running on the Browser. Additionally, we also need to ensure the integrity of the Web application code, the DOM and application data. One important characteristic we wanted to enforce in our solution is that all protection mechanisms are delivered together with the Web Application code and do not rely in any particular execution environment (such as a dedicated Browser plugin). Only a standard Browser with a modern Javascript execution environment is required. This also has the very important advantage of enabling easier updates to the protection mechanisms.

An overview of the proposed solution is presented in Figure 3. The solution is inspired by techniques reviewed in Section 3, and relies on code transformations made to the application code on the server side so that it includes the protection framework. The protection framework will then be delivered along with the Web application and executed on the client. The code transformations performed also include performing integrity checks on the data exchanged. We will now briefly discuss the two main mechanisms of our framework: (i) code execution protection and (ii) integrity protection.

5.1 Code Execution Protection

This mechanism employs state-of-the-art obfuscation techniques (introduced in Section 3) to protect the code from analysis, code injection and execution manipulation. Our approach is to bundle together the Web application code with the DOM, code and data integrity check mechanisms and use the code execution protection to ensure that they are executed as a whole, without being manipulated.

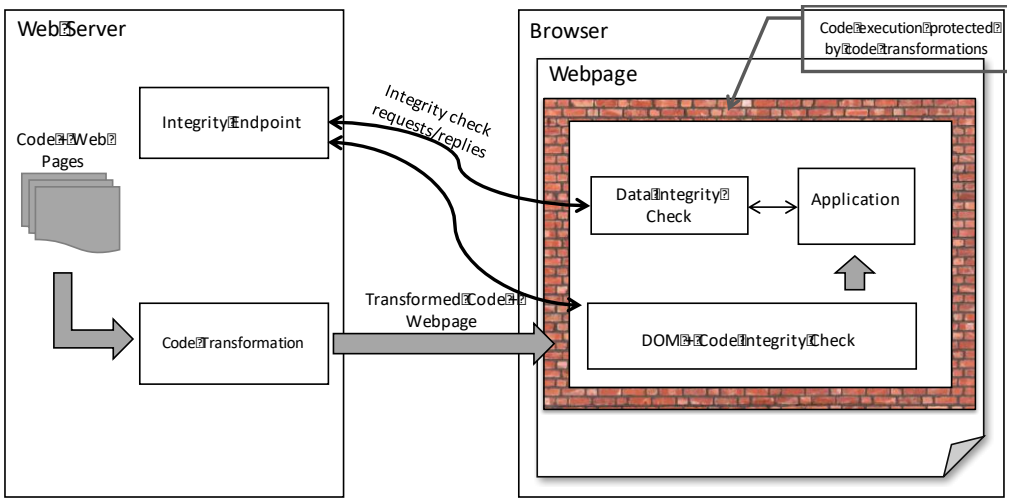

Figure 3: Overview of the Solution for Web Application Integrity.

We currently implemented several obfuscation-based techniques to impose significant barriers: (i) to application analysis (through extensive code transformations and anti-debugging traps); (ii) to application tampering (by using tamper-resistant code and tamper-detection), and (iii) to attack automation (by using diversity in the transformed code). Further details and an evaluation of this protection mechanism is ongoing work that we deem out of the scope of this first effort that aims to outline our complete Web Application integrity framework.

5.2 Integrity Protection

As indicated in our threat model, there are scenarios where data can be manipulated before reaching standard secure communication channels (such as HTTPS), therefore integrity checks must be executed by the application that is protected by the code execution mechanism discussed in Section 5.3.

The main mechanism for integrity protection is to employ a *message authentication code* (MAC). In order to create and verify MACs, the webserver and the web application running on the client need to share a session key, which we negotiate when the Web application is first delivered to the client. We do not rely on keys negotiated by TLS as these are usually not available to the Web application.

We will not go through the details of the key exchange mechanism, but our approach is to employ a well-known key exchange protocol – the Authenticated Diffie-Helman protocol (Diffie et al., 1992). This requires the client to have access to a public key of the website, and we assume this can be made available through a server certificate that the client can verify using common browser functionality. There is however one important underlying assumption: we have to trust the browser platform to perform this verification. We think this is a reasonable assumption as the browser should not allow plugins to interfere with such calls and is within our threat model.

The session key established can then be used for DOM, code and data MAC-based integrity checks as discussed in the following subsections. On the server side, we need to also compute these MACs (including a nonce to avoid replay attacks) and send them to the Web Application code running on the client. This is the task of the Integrity Endpoint depicted in Figure 3.

5.2.1 Dom Integrity Check

The DOM performs as an interface between JavaScript and the real document to allow the creation of dynamic webpages and recent security attacks targeted the DOM rather than the webpage itself (Gupta and Gupta, 2017), therefore it is important to also check the integrity of the webpage's DOM. To do this, we perform integrity checks of the DOM similar to previous work (Li et al., 2009), using a one-way hash function to compute a fingerprint of the document on the webserver and then verify this fingerprint on the client. One important difficulty to overcome is that modern Web Applications make dynamic changes to the DOM, and it is hard to distinguish the legitimacy of the changes. Our current mechanism performs static checks to selected sections of the DOM, but a more sophisticated mechanism is needed in general and we leave this for future ongoing work.

5.2.2 Code Integrity

Before the application code itself is executed, it needs to be checked for its integrity and this can be trivially done using the already established session key and verifying the MAC and nonce computed on the client with the ones sent by the webserver.

5.2.3 Data Integrity

During execution, the Web application might request data from the webserver. Our approach is to, during the code transformation phase, scan all calls that result in these data exchanges (such as calls to *XMLHttpRequest()* in JavaScript) and inject the logic necessary to perform integrity checks on these data (i.e. generate and verify the MACs and nonces). In this way, we can also guarantee the integrity of the data at the Web application being executed under the code protection mechanism.

6 CONCLUSION

We have presented a framework, inspired by existing building blocks, which delineates a possible future for Web application integrity protection. Our framework relies heavily on an obfuscation-based code protection mechanism, which enforces a trust boundary inside the browser. In this work, we focus on outlining this complete Web Application integrity framework.

As discussed, WoT applications are set to become omnipresent, and our framework becomes even more relevant under this assumption. Supporting different types of devices (interoperability), with different capabilities is one important aspect to be addressed by our implementation. We note however that there are already very capable platforms for WoT applications (Sin and Shin, 2016). Proof-of-concept implementation and performance evaluation (e.g. evaluating overhead introduced by our code transformations) of our proposed framework are left as a future work.

REFERENCES

Atzori, L., Iera, A., and Morabito, G. (2010). The internet of things: A survey. *Computer networks, 54(15), 2787-2805.*

Borgolte, K., Kruegel, C., and Vigna, G. (2013). Delta: Automatic Identification of Unknown Web-based Infection Campaigns. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and#38; Communications Security (pp. 109–120). New York, NY, USA: ACM.* https://doi.org/10.1145/2508859.2516725

Chang, H., Atalla, M. J. (2001). Protecting Software Code by Guards. *ACM Workshop on Digital Rights Management.*

Chow, S., Eisen, P., Johnson, H., and Van Oorschot, P. C. (2002, August). White-box cryptography and an AES implementation. In *International Workshop on Selected Areas in Cryptography (pp. 250-270). Springer Berlin Heidelberg.*

Collberg, C. S., and Thomborson, C. (2002). Watermarking, Tamper-proofing, and Obfuscation: Tools for Software Protection. *IEEE Trans. Softw. Eng., 28(8),* 735–746. https://doi.org/10.1109/TSE.2002.1027797

Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., … Hiser, J. (2006). N-variant Systems: A Secretless Framework for Security Through Diversity. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15. Berkeley, CA, USA: USENIX Association.*

Cylon.js. (2017, October). Retrieved from https://cylonjs.com/

Diffie, W., Van Oorschot, P. C., and Wiener, M. J. (1992). Authentication and authenticated key exchanges. *Designs, Codes and cryptography, 2(2), 107-125.*

Garg, S., Gentry, C., Halevi, S., Raykova, M., Sahai, A., and Waters, B. (2013, October). Candidate indistinguishability obfuscation and functional encryption for all circuits. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on (pp. 40-49).*

Göktas, E., Athanasopoulos, E., Bos, H., and Portokalidis, G. (2014). Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy (pp. 575–589). IEEE.*

Gupta, S. and Gupta, B. B., 2017. Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art. *International Journal of System Assurance Engineering and Management, 8(1), pp.512-530.*

Hallgren, P. A., Mauritzson, D. T., and Sabelfeld, A. (2013). GlassTube: A Lightweight Approach to Web Application Integrity. In Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (pp. 71–82).

Huang, Y., Stavrou, A., Ghosh, A. K., and Jajodia, S. (2008). Efficiently Tracking Application Interactions Using Lightweight Virtualization. *In Proceedings of the 1st ACM Workshop on Virtual Machine Security (pp. 19–28). New York, NY, USA: ACM.* https://doi.org/10.1145/1456482.1456486

IoT.js. (2017, October). Retrieved from http://iotjs.net/

Kapravelos, A., Grier, C., Chachra, N., Kruegel, C., Vigna, G., and Paxson, V. (2014). Hulk: Eliciting Malicious Behavior in Browser Extensions. In *Proceedings of the 23rd USENIX Conference on Security Symposium (pp. 641–654). Berkeley, CA, USA: USENIX Association.* Retrieved from http://dl.acm.org/citation.cfm?id=2671225.2671266

Karapanos, N., Filios, A., Popa, R. A., and Capkun, S. (2016). Verena: End-to-End Integrity Protection for Web Applications (pp. 895–913). *IEEE*. https://doi.org/10.1109/SP.2016.58

Larsen, P., Homescu, A., Brunthaler, S., and Franz, M. (2014). SoK: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy (pp. 276–291).*

Li, B., Li, W., Chen, Y.-Y., Jiang, D.-D., and Cui, Y.-Z. HTML integrity authentication based on fragile digital watermarking (pp. 322–325). *IEEE*. https://doi.org/10.1109/GRC.2009.5255107.

Nava, E. V., and Lindsay, D. (2009). Our favorite xss filters and how to attack them. *BlackHat USA, August.*

Nikiforakis, N., Invernizzi, L., Kapravelos, A., Van Acker, S., Joosen, W., Kruegel, C., … Vigna, G. (2012). You Are What You Include: Large-scale Evaluation of Remote Javascript Inclusions. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (pp. 736–747). New York, NY, USA: ACM.* https://doi.org/10.1145/2382196.2382274

Pappas, V., Polychronakis, M., and Keromytis, A. D. (2013). Transparent ROP exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13) (pp. 447–462).*

Pi.js. (2017, October). Retrieved from http://pijs.io/

Ravikumar, S., and Subramaniam, C. (2016). A Survey on Different Software Safety Hazard Analysis and Techniques in Safety Critical Systems.

Raggett, D. (2015). The Web of Things: Challenges and Opportunities. *Computer, 48(5), 26–32.* https://doi.org/10.1109/MC.2015.149

Schneier, B., "The Internet of Things Is Wildly Insecure — and Often Unpatchable", *Wired Magazine, January 2014.*

Sin, D., and Shin, D. (2016). Performance and Resource Analysis on the JavaScript Runtime for IoT Devices. In *International Conference on Computational Science and Its Applications (pp. 602–609). Springer*.

The Tessel Board. (2017, October). Retrieved from https://tessel.io/

Zouganeli, E., and Svinnset, I. E. (2009). Connected objects and the Internet of things: A paradigm shift (pp. 1–4).

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS