Athens Journal of Technology and Engineering - Volume 5, Issue 1 – Pages 7-18

An Architecture for Reliable Industry 4.0 Appliances

By Till Hänisch[*]

Industry 4.0, or the Internet of Industrial Things, means interconnected machines and devices in a heterogeneous environment. These systems have much longer lifecycles than the normal IT ecosystems we are used to in enterprise. It is difficult to keep these systems secure for an extended period of time. While minor malfunctions may be acceptable, software bugs might lead to security problems, which cannot be ignored, since they will have consequences in the real world. Because of this, it is important to keep the number of bugs as low as possible and to limit the damage of the remaining ones. Today's method of keeping systems (like operating systems) secure is to patch them permanently to close all discovered bugs. The necessity to patch on a regular basis combined with the long lifespan of the components creates serious interoperability issues. To handle these problems with acceptable effort while keeping a high level of security, they must be addressed on different levels such as the operating system, the network architecture, composition of services, and programming. The key to a successful long-term perspective of such a system is a flexible architecture that allows maintenance and extensibility in a controlled environment, while preserving the integrity of the system. In this paper, a flexible architecture is described, which isolates critical components and allows the substitution of components without compromising the system in case of failure. It consists of clearly separated services with well-defined interfaces that can be enforced by the runtime system.

Keywords: *Functional Programming, Internet of Things, Microservices, Security, Unikernel.*

Introduction

There are many aspects of IT-Security in the Internet of Things, such as securing the transmission of data over the internet to provide confidentiality and integrity, or using authentication and authorization to provide integrity and availability (Babar et al. 2011, Chethan et al. 2016, Bouij-Pasquier et al. 2015). This paper focuses on a different aspect: How can an application itself be made as secure as possible?

In other words, how do we construct an application in a way that it contains as few errors as possible and how do we limit the damage of the remaining ones? Since errors are not only a safety problem (functional security), but also might lead to bugs that can be exploited, this is a question highly relevant to the security of a system. This is especially important as such applications typically have a lifespan of decades, much longer than typical computer systems in the enterprise.

Of course this is a principal problem not only of the application but also of the underlying platforms and operating systems (maybe even the underlying hardware, if microcode is considered). However, there are solutions for these platform issues, mainly through trusted update mechanisms. The supplier of the

[*] Professor, DHBW Heidenheim, Germany.

https://doi.org/10.30958/ajte.5-1-1 doi=10.30958/ajte.5-1-1

platform is supposed to deliver regular updates to maintain the security of the underlying system. While this is typically not true for today's systems, this is more a cost problem than a fundamental problem. If the supplier of the platform is liable for security problems, he will have a strong interest in keeping the systems secure. This is feasible because there are possibly a large number of his systems in the market, allowing the cost to be shared between many customers.

With the application itself, it is a different and more difficult problem. Individual applications will probably not be produced and deployed in such a large number, and so regular updates can be guaranteed for a long time. The supplier may even go out of business. There has to be at least an additional mechanism to keep the system secure on this level. If the application is designed with a well-defined interface and does not contain any errors ("bugs") in the code, this would be a large step in the right direction. Of course, this is not easy to achieve.

Governmental organizations, like the US administration (Detsch, 2016) or the European Union (ENISA, 2016), start to require something like security by design. How could this be implemented?

This paper describes some techniques to help with these efforts. Basically, these are the same techniques coming up in large-scale web development today (2016): microservices, containers and tools from the functional programming ecosystem. The contribution of this paper is to transfer these concepts from the worldwide large-scale systems of the Internet to applications in small Subnets of Things (Machina, 2013).

Code Level

Today, writing programs does not mean starting from scratch, selecting algorithms and coding them in your language of choice, but rather tying libraries together to fit your needs. This is especially true for IoT applications, be it low level coding on an Arduino-like board to collect sensor data or synchronize actuators, or be it high level on a pi-like (or even more complex) system.

Security-wise, such a program can be made error-free, if we assume that the libraries are error-free and the glue code making up the application itself is error-free. However, the assumption of error-free libraries is wrong. The implications of this are discussed in another section (Architecture). It can be assumed that the quality of the library code is better than the quality of the application code, because it is used more often, for a longer time, and is open source in most cases. Although there are no formal studies about the higher security level of open source code on a large enough sample to be considered applicable in general, it is a widely accepted "fact" in IT, backed by arguments of experts in this field (Schneier, 2011). The rest of this article will focus on the user written glue code.

Connecting libraries usually means moving data around between function or method calls. To do this without having to write large amounts of boilerplate code, for example to convert between different data types, flexibility is key. Statically typed languages like Java do not excel at these tasks, because their

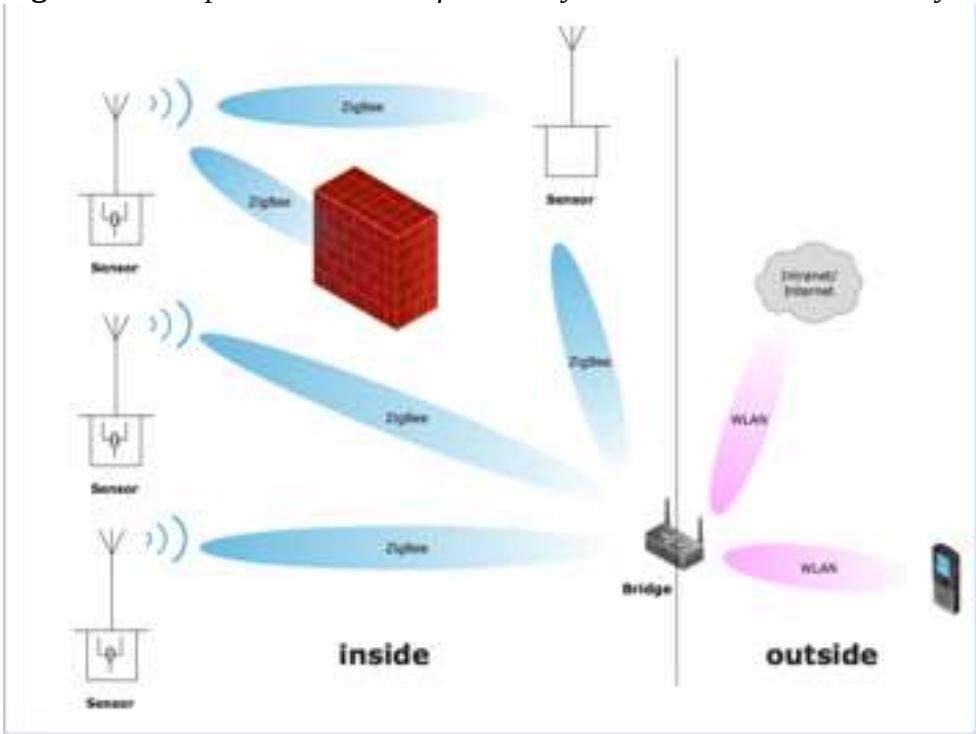
static type system enforces many manual conversions, which not only require more typing, but also result in confusing code where the important logic of the program is hidden in the details of language specific formalism.

This is one of the reasons (Ousterhout, 1998) why dynamically typed languages, often called scripting languages, became more and more important. This trend can be seen especially with the rise of JavaScript in web development. JavaScript is a weakly and dynamically typed language, which makes simple things easy, but leads to severe problems with bugs. For two years, (https://www.google.de/trends/explore?q=TypeScript) there has been a trend to languages like TypeScript or Elm, which provide more reliability for finished applications by using stronger type systems for browser based development.

With these new languages, the old problems of statically typed languages - such as more code, less flexibility, and so on (Ousterhout, 1998) - show up again. The interesting question is if there is a way to have both flexibility (meaning less work when writing programs) of scripting languages and reliability (meaning better readability and more type checking by the compiler) of statically typed languages.

Code generators and domain specific languages (DSL) are established ways to create an intermediate abstraction level to reduce complexity and code size by creating programs nearer to the user domain, and let as well a computer program do the job of transforming a higher level specification to a machine readable form.

Figure 1. *Example Architecture of an IoT System with a Central Gateway*



This idea can be applied to IoT applications as well, at least to some architectures. Figure 1 shows a typical architecture of an IOT application: The sensor nodes, which collect data from their sensors, send them to a central gateway node (called "Bridge" in this figure) using an appropriate protocol, ZigBee in this case. The central gateway collects the data from the sensors and makes them available to the clients. A different network protocol is used here in order to make it easy for the clients to gain access to the data, WLAN in this case.

Haenisch (2016) shows that applications with an architecture like the one in Figure 1 can be described in two parts: sensor nodes, which do the low level (maybe real time) handling of sensors and actuators, and gateway nodes, which process the messages to and from the sensor nodes. The function of the gateway nodes can be described as message processing, which works according to rules and creates new message streams. This process can easily be described in a DSL.

With the sensor nodes, this becomes a bit more complicated since it is not clear on which application domain the DSL should focus. The possible applications and combinations of sensors, as well as the methods of interaction between them, are just so large that it seems difficult to specify a DSL that will capture all or even a large group of applications. Because of this, a different approach is needed.

There are a number of approaches that attempt to solve this problem, such as Functional Reactive Programming (FRP) for IoT nodes, for example Frp-arduino (https://github.com/frp-arduino) or Juniper (http://www.juniper-lang.org/).

Juniper (Helbling and Guyer, 2016) is a programming language for the arduino using FRP. Frp-arduino (Frp-arduino, 2016) is a Haskell library and precompiler that claims to provide "Arduino programming without the hassle of C". Both of these have the problem that although they provide a library with common idioms used in more or less simple applications, they force the user to use languages like Haskell, which is alien to the typical developer of an embedded system. Frp-arduino uses the concept of an event stream like the techniques described above.

Applications like this are especially well suited to using functional languages like Erlang (Sivieri et al., 2012). There are case studies (Haenisch, 2016) that support this assumption.

All of these techniques aim to reduce the number of bugs in the application code, but this approach will probably never lead to a completely bug-free program. The system as a whole should be as resilient as possible even if part of the application fails or is hacked. This must be addressed at the architectural level.

Architecture

Basically there are two ways to increase the security of an application like the control system of a machine: either by using a monolithic architecture, which is as simple as possible and has a small attack surface, or by composing

the system of independent microservices where the security of the system is distributed across the components.

Monoliths and Microservices

A monolithic architecture consists of a single unit that contains the complete application logic. Typical enterprise applications are often built this way: they consist of a client-side user interface and a server-side application layer, which accesses a database. For every change in the application logic, the system has to be rebuilt and deployed, a process that requires much care.

Though it appears simple at first, this architectural style has some disadvantages. Every change to a small part of the application requires building and deploying the whole system. From a security perspective, there is a similar problem. Every security problem for a small part of the application results in a problem for the whole application. The application is running as a single process in a single memory space, so all of the application runs in the same security context. This means that every single bug in a small part of the system gives a possible attacker complete control of the whole system. Certainly this is not what we want to have, especially in an Internet of Things scenario.

In micro-service architecture, the system is built from a group of lightweight components, which use loose coupling and well-defined interfaces for communication. At the time of this writing (2016), this is a well-accepted development style in web development, mainly because of good fit to successful organizational patterns, easier testing, and easy continuous deployment (Fowler, 2014).

Another important property of micro-service architectures is that they support or even enforce designs for failure. Because service calls over a network can fail, all of the components of the system must be well aware of this fact and take appropriate counter-measures (Fowler, 2014). This has consequences for the security of the system.

Since the services are isolated processes (maybe even running on separate operating systems, see next section), separated by well-defined interfaces, a security problem in one service will not be able to compromise other services if and only if access to services is managed to be secure.

Of course, if done wrong, a micro-service architecture might actually decrease system security by increasing the attack surface through additional interfaces, communication channels, and data sources. Done right, it might lead to gains in resilience and increases in security.

Unikernels

Making a platform secure is a difficult task. One lesson learned over the years is that complexity is the enemy of security (Geer, 2008). A complete operating system with all its features contains a number of undiscovered bugs and associated exploits. Because of that, it is common wisdom to uninstall and/or shut down all unneeded services to reduce complexity.

Unikernels take this principle to the extreme to increase security (Kurth, 2015a). Unikernels are specialized OS kernels built using a library OS (Madhavapeddy and Scot, 2014). The resulting image contains only the code, which is required by the application, compared to a standard operating system, where the kernel contains all the code any application might need. The current linux kernel (4.2) contains some 15 million lines of code, each of which might contain a bug. A typical Unikernel image in MirageOS, a popular unikernel implementation, contains some 50 thousand (Madhavapeddy and Scot, 2014).

In addition, because every kernel is different, many conventional exploit schemes do not work. Everything based on fixed addresses, like stack overflow exploits, must be tailored to the exact kernel running on the system. It is quite easy to make every kernel a little bit different.

Another important feature of unikernels is their small size. This reduces boot up time to the order of tens of milliseconds (Madhavapeddy et al., 2015), an important feature in real time and/or low power applications (Unikernel, 2016).

This explains why unikernels get more and more attention as a method of building secure applications. This is not really a research topic, and there is no accessible systematic research on this topic, at least as known to the author. However, at high-tech conferences like CodeMesh (www.codemesh.io), it is a hot topic (Garnaes, 2016; Smith, 2015) for the combination of unikernels with micro-services.

Using containers or using unikernels are both techniques that are very promising for increasing performance and security of application stacks. This is true for the normal IT-world with its application servers and micro-services in web applications, and it also looks very promising as an architectural model for embedded components. "Will enterprises deploy a mix of VMs, unikernels and containers? Or will unikernels eventually go mainstream and replace containers?" (Kurth, 2015b).

Using unikernels enables superior isolation between the parts of a system in a machine, while building up on the rich hypervisor ecosystem. Containers like Docker on the other hand provide a much easier system for deploying components.

From a security perspective, the easier solution is the better. Thus, unikernels running on a widely used lightweight hypervisor like Xen (https://www.xenproject.org/) seem to be the way to go, especially if platforms like LING (http://erlangonxen.org/) are used, which run an application platform, in this case Erlang/ OTP directly on Xen (Sivieri et al., 2012).

Proposed Architecture

Based on the interviews described in Haenisch and Rogge (2017), we found to our surprise that large companies do not use application layer firewalls to isolate single machines in their production environments. The reason for this is that large companies with distributed production sites all over the world and centralized IT security departments are not able to maintain the configuration

of these firewalls. The communication and administrational overhead required is too involved.

Figure 2. *Illustration of the Dependency of the Required Effort from the Number of Systems for Layer 7 Firewalls (AL Firewall) and an Intrusion Detection System (IDS)*

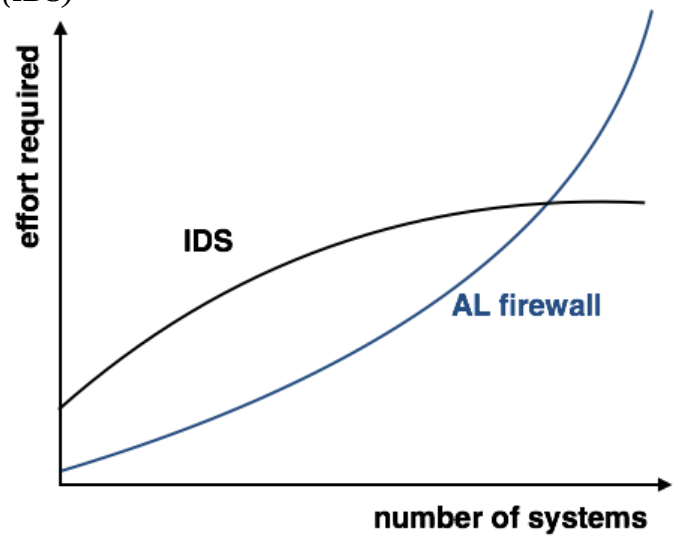
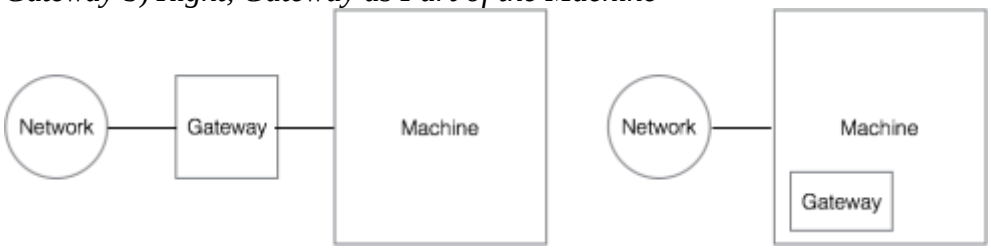


Figure 2 illustrates how a large number of systems increases exponentially the complexity needed for maintaining their configuration. Because of this, large companies have no other choice but using different mechanisms like segmentation and intrusion detection systems. This is completely different for small to medium enterprises with only a small number of machines. The effort for configuring one or a few application layer firewalls is much lower than every other technique. For these companies this is the recommended solution. This architecture, shown in Figure 3a, is also commonly used with legacy systems like mainframes, which cannot be updated, similar to the problem with production machines with outdated operating systems and application programs.

Figure 3. *Isolating a Machine from the Company Network* a) *Left, Separate Gateway* b) *Right, Gateway as Part of the Machine*



The short term solution for all of these problems is a separate application layer firewall or gateway between the single production machine and the company network. While it is possible for every company to use a standard platform and configure the firewall with its own resources, it is questionable if this is the

best option. Setting up and maintaining a secure configuration of a network gateway requires special knowledge that is typically not available in smaller companies. Even for large companies this is not their primary business, so a solution where this is outsourced would be preferable. The question remains; who should initiate and pay for this work.

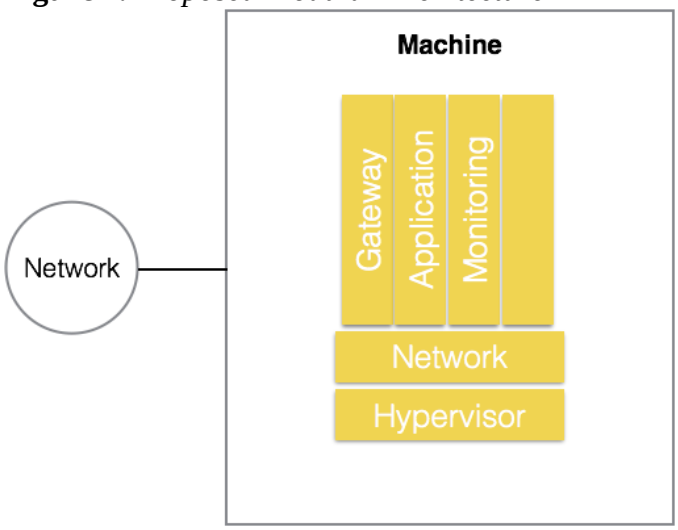
Probably not every single owner of a machine can initiate this, because this would require building and especially maintaining a large number of similar but different devices. While this heterogeneity might actually increase security, economically this does not make sense. It would be much easier and cheaper if the supplier of the machine delivers the gateway and guarantees maintenance. From a conceptual point of view this means that the gateway is part of the machine (see Figure 3b).

Again the question remains if the machine manufacturer should develop this gateway on its own or outsource to a specialized company. The same argument as above applies here: the know-how required to build a secure platform and the manpower required to maintain it is not the primary business of a machine manufacturer. Thus, a specialized partner should take over this work.

In this scenario, one question remains: Who takes the residual risk in case of a malfunction or failure of the gateway? This would result in a production loss at the end user. The supplier of the machine does not want to take this risk, and getting rid of this liability is the motivator for delivering the gateway as part of the machine. The maintainer of the gateway platform typically is a small, highly specialized company, so it is questionable whether it can handle the cost of the production loss, which might increase with a larger customer base.

To handle this case, a third partner is required for insurance. The business of insurance is to take residual risk, so this is a natural fit. Additionally, this leads to a win-win-situation: The insurer is interested in paying as rarely as possible. That means the insurer is interested in having a gateway that is as secure as possible. This means there is interest to pay for increased security of the gateway. That implies the organizational structure, a partnership between the machine manufacturer and a joint venture between a security company and an insurer, which offers a single product, an insurance against security related production loss which contains initial installation and maintenance of a gateway to reduce the probability of this risk occurring.

Figure 4. *Proposed Modular Architecture*



While the gateway may be seen as a black box, it makes sense to use an architecture using the techniques described above (Figure 4). Building the functionality from several micro-services as components allows flexible configuration and maintenance by the machine manufacturer and the security partner. Different machines can share functionality, which allows developing these shared components to higher standards of software engineering and security. The diversity of the services keeps security high, using isolation techniques such as containers or unikernels. Machine specific functionality can be implemented with the techniques described in chapter "Code Level", like using functional programming languages and domain-specific languages to reduce the number of errors in these parts of the system as much as possible. Running small containers, like Erlang modules, directly on the hypervisor especially reduces the attack surface as much as possible.

Discussion

There are a number of commercial attempts to run platforms like Erlang/beam on small hardware for embedded systems coming up. The idea here is to use functional programming and especially the Erlang ecosystem (OTP) to get the benefit of reliable crash and change resistant application architectures in IoT, especially industrial IoT applications (Sivieri et al., 2012).

In building embedded systems there is a long tradition of using graphical models of the application code. Especially the graphical (and configuration based) connection of existing components is a well-established technique, the best-known example of which is LabView. This is a similar approach as using DSLs or code generators, but is usually tied to a specific development environment from a specific vendor.

All of those try to address the problem of minimizing errors (and the difficulty of building the code) in the application glue code that connects the prefabricated

components. However, all of them use a proprietary language (text or graphical) that limits the user (the developer in this case) to certain architecture and forces them to learn a new language. This leads to a vendor lock-in and tends to limit flexibility. A vendor-neutral open source solution has many benefits, especially considering the long lifespan of these systems.

There are a number of reference architectures for IoT applications (Babar et al., 2011; Bouij-Pasquier et al., 2015). It remains to be seen if general models of IoT applications can be developed. This would enable a common application model and a common language for the implementation of IoT applications on an abstract, less error prone level than today's environments. Using micro-services increases complexity by using a distributed system with additional interfaces, but this can increase security in the whole system (Mulesoft, n.d.).

Conclusions

Holding security on a high level for many years is a central problem in the development of the Internet of Things and especially in its industrial application. Many roads lead to Rome, but this paper suggests a specific solution to this problem. Reduce the number of bugs in the application code by using an appropriate representation, and use a modular architecture with an attack surface as small as possible to limit the impact of a security problem and to allow to adapt to future needs without changing the whole system. These are current trends in web development and they apply in this area too.

References

Babar, S. et al. (2011) Proposed Embedded Security Framework for Internet of Things (IoT). Conference paper: 2nd IEEE International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology (Wireless VITAE), 2011, Chennai, India. Volume: Proceedings of Wireless VITAE 2011.

Bouij-Pasquier I. et al. (2015) A Security Framework for Internet of Things. In Lecture Notes in Computer Science 9476, Springer 2015, ISBN 978-3-319-26822-4.

Chethan C. et al. (2016) Applications and Challenges of Internet-of-Things- A Survey, International Journal for Scientific Research and development Volume 3, Issue 11, 2016.

Detsch, J. (2016) Should companies be held liable for software flaws? http://bit.ly/2gQEBYd.

ENISA (2016) ENISA urges 'security by design' for EU digitization. http://bit.ly/2zM4FA6.

Fowler, M. (2014) Microservices, a definition of this new architectural term. http://bit.ly/2jGMDXV.

Frp-arduino (2016). https://github.com/frp-arduino/frp-arduino.

Garnæs, A. (2016) Gossiping Unikernels, CodeMesh 2016, http://bit.ly/2iKGUxj.

Geer, D. E. Jr. (2008) Complexity Is the Enemy. IEEE Security & Privacy, November/December, p. 88.

Hänisch, T. (2016) A Case Study on Using Functional Programming for Internet of Things Applications, Athens Journal of Technology & Engineering, Vol 3, Issue 1.

Hänisch, T. and Rogge, S. (2017) IT-Sicherheit in der Industrie 4.0 [IT-Security in the Industry 4.0]. In Andelfinger, Hänisch (Hrsg.), Industrie 4.0 – wie cyber-physische Systeme die Arbeitswelt verändern, Springer.

Helbling, C. and Guyer, S. Z. (2016) Juniper: A Functional Reactive Programming Language for the Arduino. Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design Pages 8-16, Nara, Japan. September 24 - 24, 2016.

Kurth, L. (2015a) Why Unikernels Can Improve Internet Security. linux.com, 1.4.2015, http://bit.ly/2hZRtNv.

Kurth, L. (2015b) How Early Adopters Are Using Unikernels - With and Without Containers. linux.com, 30.3.2015, http://bit.ly/2m9aYYi.

Machina Research (2016) Big Data in M2M: Tipping Points and Subnets of Things. http://bit.ly/2jg6BJO.

Madhavapeddy, A. and Scot, D. J. (2014) Unikernels: The Rise of the Virtual Library Operating System, Communications of the ACM. Vol 57, No. 1.

Madhavapeddy, A. et al. (2015) Jitsu: Just-In-Time Summoning of Unikernels. Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation.

Mulesoft (n.d.) Microservices and Security: Increasing security by increasing surface area. http://bit.ly/2zvZXFE.

Ousterhout, J. K. (1988) Scripting: Higher-Level Programming for the 21st Century. Computer, 31(3): 23-30.

Schneier, B. (2011) Open-Source Software Feels Insecure, Schneier on Security. http:// bit.ly/2ypqh0F.

Sivieri, A., Mottola, L., Cugola, G. (2012) Drop the Phone and Talk to the Physical World: Programming the Internet of Things with Erlang SESENA '12. Proceedings of the Third International Workshop on Software Engineering for Sensor Network Applications, Pages 8-14.

Smith, G. (2015) Rainbows and Unikernels. CityCode Chicago. http://bit.ly/2i01PwD.

Unikernel16 (2016) unikernel.org.