Software Security Considerations for IoT

Aki Koivu, Lauri Koivunen, Shohreh Hosseinzadeh,
Samuel Laurén, Sami Hyrynsalmi, Sampsa Rauti and Ville Leppänen
University of Turku
Turku, Finland
Email: {aki.i.koivu, lamkoi, shohos, samuel.lauren, sthyry, sjprau, ville.leppanen}@utu.fi

***Abstract*—Internet of Things (IoT) is a swiftly growing technology and business domain that is expected to revolutionize the modern trade. Nonetheless, shortcomings in security are common in this new domain and security issues are the Achilles' heel of the new technology. In this study, we analyze different security solutions for IoT devices and propose suitable techniques for further analysis. The aim of this study is to provide guidance on implementing security solutions for both existing and coming devices of Internet of Things, by providing analysis and defining the Complexity of Implementation score for each solution.**

***Index Terms*—Internet of Things, security, software security**

I. Introduction

Internet of Things (IoT) is a network of connected devices [1]—or 'things'—that collect and share information in order to create a more automated environment. These 'things' can be in the form of sensors, actuators, software, and network connectives (e.g., a gateway that connects home light bulbs to the Internet). IoT is considered as the third wave of IT-driven competition [2]. The number of applications for IoT devices are growing rapidly as the technology is becoming more popular. Therefore, security of IoT networks and devices is of paramount importance.

Prior research shows that security in IoT still has room for improvement [3], and traditional security measures developed for enterprise and desktop computing are not compatible with this environment [4]. Confidentiality, integrity, and availability of information security hold for IoT as well, but the implementations addressing these aspects must be modular and lightweight. Security requirements for IoT devices focus on communication security, data protection and physical protection [5].

There exists a wide range of heterogeneous components with different processing capabilities and capacities [6]. Some devices are equipped with 32-bit processors and have memory hierarchies like a smart phone, while some other have only a 8-bit micro-controller [7]. Moreover, many of the participating devices are constrained in memory, storage, and computational power. As an example, a networked door lock is a fairly low capacity device that has just enough processing power to have the essential functionality, and is not capable of performing complicated computations. To this end, a security measure needs to be compatible with even most limited devices of the network and certainly, not all security measures may apply to all IoT devices. Therefore, it is important to identify security measures feasible for implementation in the IoT environment.

In this study, we research the complexity of implementing security measures into IoT devices, with the goal of identifying conveniently implementable and lightweight security measures. We list various security solutions that either have already been implemented in embedded IoT devices, or are experimental in nature and have the potential to be implemented. Every solution is introduced and our findings are presented. The remaining of the study is structured as follows: Section II surveys the relevant studies. Section III defines our scope of study. Section IV outlines each security solution. Finally, Section V provides conclusions based on the research and possible future work directions.

II. Previous studies

There are many different security solutions proposed for various domains, and providing security at different levels, i.e., hardware, software, and network. Some of these solutions are applicable to IoT environment as well, while, some other are not applicable due to the limitations of the participating devices in these networks. For instance, a typical operating system security method, Address Space Layout Randomization (ASLR), can not usually be applied on IoT devices. This is because the IoT devices routinely lack the required hardware (such as a Memory Management Unit), or software structures (such as relocatable or loadable libraries).

There is a noticeable amount of research on IoT focusing on different aspects. There exists a systemic IoT security study by Riahi et al. [8], which can be used as a general guideline for implementing overall IoT security. There is also a targeted study on security considerations for home IoT by Han et al. [9], which discusses the specific requirements of implementation. Connectivity has also been researched, by considering the cloud aspect of IoT security [10]. Alqassem et al. [11] authored a paper on the requirements for IoT security on an abstract level, while a security analysis of existing IoT devices by Wurm et al. [12] provides ways to exploit existing devices. This study is a good ground for researching ways to defend against the outlined attacks. Also, a study by Hosseinzadeh et al. [13] gives an overview of obfuscation and diversification as security strengthening solutions for IoT.

III. Scope of study

A. Complexity of Implementation

In device manufacturing, production cycles are rapid and existing code or software is often recycled. Because of this, additions to the used code should be modular and not require rewriting an extensive amount of the original software. Adding security measures as new blocks of code to the software should also obey this requirement.

In our study, we focused on finding security solutions that evaluated to be the most practical to implement as new features of existing software. For measuring practicality of the solutions, we introduce the Complexity of Implementation score (COI):

$$COI = 1 + C1 + C2 + \dots$$

Each input value is a binary 1 or 0 score based on a criteria that contributes to the complexity of the solution. The goal is to estimate how difficult it would be to implement a certain solution to an IoT system, based on the existence of known implementations and the extent of current knowledge of the subject based on research available, the estimated amount of software code needed to be modified and the applicability with different architectures. In our case, the COI score ranges from one to four, where higher score indicates a possibly more complex and harder solution to implement. No solution is trivial to implement, which is why we set the base score at one. We use the following three criteria to define our COI scores:

- C1 Instead of just adding more code, code rewriting in existing code is required for integrating the solution.
- C2 Major architectural changes are needed in existing code for integrating the solution.
- C3 Existing implementations of the solutions were not found, further research required.

The scoring is noticeably coarse, but provides a common metric for evaluating the possible complexities of a security solution.

B. Limitations

We excluded software development methodologies from our study. Most of them can not be applied as an afterthought by their very definition, such as Test-Driven Development (TDD), which we want to avoid. In addition, TDD should trivially apply to any newly written software in the context of IoT and is therefore, not worth studying in this context either. Introducing security awareness to code development has already been studied in the literature [14] and we believe this literature should be used when new software is to be written regardless of whether it is targeted towards IoT or not.

We also exclude static analysis from our study. Static analysis of software is a useful tool to reduce bugs in the code and also increase software security [15]. Large organizations, like NASA, are both researching and using static analysis in their workflow [16]. This alone makes us believe it should contribute to the security and correctness of IoT devices software the same way it does for other software. We believe this is even more relevant for IoT due to the minimalism of many IoT systems allowing more fine-grained analyses to be performed compared to bigger systems.

In our study, for conducting research we used an STM32L1 microcontroller-based IoT device, the Thingsee One [17]. The device is equipped with an ARM Cortex-M3 processor and an IoT-oriented operating system, NuttX OS. Operating systems for IoT purposes are designed to be portable, because there is a great deal of diversity among IoT hardware. Currently, embedded IoT operating systems like RIOT and FreeRTOS showcase support for various microcontrollers, but are often implemented only for a finite group of devices. While we experimented with a group of operating systems, our main focus is on NuttX OS.

C. Threat analysis

It is essential for any security measure to define what threats it is defending against. By definition, IoT devices are connected, which makes network attacks major sources of threats, but we also believe that as IoT devices become more commonplace, physical security also becomes important. Otherwise, an attacker could, for example, walk up to an internet enabled coffee machine and re-flash its firmware, which could be used to do anything from brewing coffee for free to providing reverse proxy access to internal networks. To be more specific, we want to counter software bugs and vulnerable software that could lead to a system compromise, which can come from both the network or from a malicious user having a physical access to the system.

IV. Security Solutions

A. Memory protection

While researching IoT operating systems and the IoT devices they are used on, we noticed many of the devices do not have a Memory Management Unit (MMU) subsystem and as a consequence many of the operating systems also lack any support for MMU. Traditional memory protection mechanisms, such as ASLR [18] cannot therefore be used. A simpler subsystem support called Memory Protection Unit (MPU) does exist in many of the devices and operating systems. It can provide an alternative design for userspace separation to provide some level of integrity and memory protection. The operating system ARM mbed even uses it to provide a similar level of protection afforded usually only by MMU [19]. This feature is limited to ARM's processors only, unfortunately.

We researched the commercial but mostly open-source Thingsee One device, to see whether we can enable the operating system's separation of kernel and userspace, which was not used by default. This would allow us to enable memory protection. The devices processor, ARM Cortex-M3, has the capability to protect memory [20]. According to our findings, it is challenging to integrate kernel and userspace as an afterthought, as this requires a conscious choice to separate components of the system as a whole. Extra systems added on top of the operating system for the device were calling the kernel features directly and would therefore require at least some rewriting to function in the more secure operating mode of the operating system used.

Even if one can not enable the operating system's protection mechanisms for application and/or kernel protection, or there is no real userspace, memory protection units can still often be used partially for securing at least parts of the system. For example, on ARM processors, the interrupt vectors are often located at the start of the address space [21]. A problem arises because null memory pointers could provide an attacker a way to write code in place of an interrupt vector and through that redirect execution easily to a malicious code [21]. The vulnerability can be avoided by using the otherwise unused memory protection unit of the system. Some invalid memory pointers may not point right at the start of the address space, but could instead be dangling elsewhere in the memory space. They could even be just slightly offset from the beginning of the address space, which could be used as another attack vector. So, protecting just interrupt vectors at the start of address space is not enough, but is a good start for efforts hardening IoT against potential vulnerabilities and understanding how MPU can be used.

We give memory protection 2 in our COI scoring as it requires specificity from hardware architecture in almost all cases, barring stack checks, for example. Memory protection is mostly a well established practice and therefore does not need further research to be implemented. Writing code is required, but memory protection can be added as a layer on top of everything to enforce constraints instead of requiring code modifications for the code to be made "memory-protected". There are also plenty of existing memory protection implementations in various forms [21], [22].

B. Link time reordering

A typical embedded IoT operating system is assembled from multiple pieces. These pieces, or object files, are laid out into a final address layout for a device. Since the layout order is deterministic between pieces (whose relative order does not matter), an attacker can use the knowledge of a single device instance to figure out the layout of another device instance and then use this knowledge to attack the system further. This knowledge is internal implementation details and can therefore be changed without breaking the system. We can introduce artificial diversity to the linking phase to shuffle the memory layout while preserving functionality, a form of diversification.

On our tested device, the Thingsee One, there is a binary blob, which makes many forms of security measures difficult to apply. Fortunately, the binary blob is an object file which can be shuffled with the rest of the system. This means link time reordering can potentially be used with partially closed source systems, which is desirable due to many vendor-developed opaque binary blobs containing driver code. It is also necessary to note that whatever is inside the vendor's opaque blob will naturally not get reordered internally, but the impact of this can be limited. Vendors should be made aware of this sort of security measure. This would then allow them to divide up the said binary blobs into smaller parts that could still be opaque, but their order could be shuffled within the resulting system on a finer level.

Link time reordering is scored at 2 in complexity of implementation. The technique requires only compile-time architectural changes to the linker and no code should require rewriting. The principle is straightforward, but its effectiveness may require further research. No existing implementations were found, which may indicate either the lack of research or tools.

C. Code obfuscation

Code obfuscation is often regarded as security by obscurity, but it can be used as a useful layer of deterrence on IoT if applied on per device basis [23], [24]. This is known as code diversification. Generating a generic attack that would work on all instances of the device would be made nearly unattainable with proper diversification. This can be done, for example, by instrumenting code at either source level, compiler intermediate code level or at compiled bytecode level by adding NOP (no operation) instructions [25].

Obfuscation can be a useful method even when executed in a naive way, but better tools are required for maximal effectiveness against vulnerability analysis and exploitation. This area needs further development. Since the most effective obfuscation needs to be done per-device, a further limitation of this method is that compiling becomes a bottleneck. Depending on operating system complexity, a compile pass may take a long time, but fortunately many IoT systems with included applications are often so small that this should become almost negligible. For example, the NuttX operating system compiled with a simple 'hello world' printing script does not take more than a minute to build on a recent desktop hardware. This is not always the case, though. The upcoming Brillo operating system for IoT by Google can take up to 30 minutes to finish a compile pass in the same system.

Obfuscation can often also lower performance, which can be critical on real time systems. Therefore, obfuscation might not always be applicable as a security increasing solution. Furthermore, debugging systems that have been changed per device can be harder as, for example, offsets for debugging data are different for each device. This can be alleviated by being able to compile the debugging symbols afterwards based on a seed that is normally kept secret. The seed would basically be input into the compiler and would provide a deterministic compile pass so that all debugging offsets are found.

Code obfuscation is scored at 1. There is plenty of research and existing implementations on code obfuscation. By its very definition, we do not need to change the architecture of a system to apply obfuscation. Code rewriting is also not required if it is done automatically. Nevertheless, obfuscation is a complex subject even if its base cases are simple.

D. Stack shuffling

The stack of a program holds data such as local variables and execution flow information of function calls, such as return pointer to calling function. The boundaries of variables and the location of the stack is often deterministic and known to the attacker along with how the data is going to be used in the stack, which can allow malicious actor to perform attacks using buffer overflows in stack variables. The order of the variables that are pushed into the stack can often be changed without much consequence, which stack shuffling aims to do. By modifying how the stack is being constructed per device, we can make it more difficult to perform stack-based attacks. Stack shuffling would suit well to the IoT environment for finding bugs and preventing exploitation because of its negligible performance drop. There exists at least one implementation of stack reordering on a compiler [26], but it has been made specifically for systems using the OpenBSD operating system. Using it to develop a general implementation should be possible. The security evaluation of this technique is still lacking, but as buffer overflows have been a source of many vulnerabilities [27], this approach should have positive effect on security.

This approach is scored at 2 as implementation of it for various compilers and processor architectures may not be trivial and hence require further research. As mentioned above, it has at least one existing implementation, but its effectiveness is not widely tested based on our findings. Code rewriting or architectural changes are not needed except possibly in the compiler itself.

E. System call diversification

As with link time reordering and stack shuffling, diversifying system calls needs to be done per device to be of any use [28]. Also in IoT many devices contain some type of system call interface. Many, if not all ARM processors for example, have SVC-calls (Service Calls) that pass control to the kernel code. The goal of system call diversification is to make constructing meaningful system calls difficult for an attacker. The attacker cannot then for example write into a file descriptor (a system call action) to further exploit the system.

This security solution does not apply to all IoT systems, as the systems may be so small that it does not make any sense to have system calls implemented. Due to the invasive nature of this method on the system, having to modify the fundamental building blocks of a kernel and userspace separation, it may be overly burdensome to implement this feature on some operating systems. One would face difficulties with systems, for example, if system calls were not initially enabled. All custom code calling directly into kernel would need to be rewritten, since it is a prerequisite for the system calls to have kernel and userspace separation for the system calls to make practical sense to exist. Our target system did not use the kernel separation and therefore programs that were in userspace were attempting to touch kernel data directly with separation enabled, which hindered writing our implementation greatly.

In practice, system call diversification can essentially be done by modifying the order of system calls so that each device has a unique and externally unguessable order of the possible calls [29], such as writing to a file and run executable calls. System call diversification appeared to be feasible at the very least on the NuttX operating system, which we inspected in depth. On other IoT operating systems, if the system call ordering/numbering is being hardcoded in header files instead of being generated, automatic diversification could be problematic initially, but on architectural level we can see no other big obstacles.

This security solution is scored at 3. It requires architectural changes at worst, which also implies code rewriting. There is enough research on the subject to write an implementation. Commercially the security solution is not very well known, but this may be because it is rather new or there are not enough tools or operating system level support.

F. Integrity verification

Integrity verification can take many forms. For IoT the universal Serial Bus (USB) is a technology that is very common among today's devices. The technology is also typical among embedded IoT devices, and contemporary energy-efficient USB technology is now developed for IoT [30].

USB connectivity can provide attackers plenty of different ways to attack a device. In one instance, a USB mass storage device was made to emulate a USB hub device, attaching and removing virtual devices in a specific sequence in order to manipulate host's memory heap, which ultimately lead to a buffer overflow [31]. This particular attack was made against a video game console. In certain sophisticated IoT devices, like the ThingSee One, the USB port can be used to access the device's storage or flash in new firmware. When this feature is present, malicious software can be injected to the device, for example with a USB drive. The implications of these attacks would be intensified in a high-risk environment like industrial or military systems.

Integrity verification for USB connectivity is a crucial feature for strengthened physical-layer security. The host device requires a specific identification number from the connected USB drive to grant it read and/or write privileges. Every USB drive has a manufacturer-defined device ID, composed of vendor, product and revision codes. This ID can for example be used to identify a USB drive without additional mechanics. Integrity verification for USB has been implemented for example as a part of embedded lock-down manager in Windows IoT (formerly Windows Embedded) [32], but as a lightweight solution for IoT devices without an operating system this feature can be built into the kernel of the device, by manufacturer-defined or user-defined configuration process. Apart from the perspective of USB, integrity verification can for example be seen in the accessible serial ports as requiring a password [33].

Integrity verification is scored at 2 as it requires some non-trivial architectural changes to account for certificates and validation. Some integrity verification types, such as secure boot, would require supporting hardware, which may require complex changes.

G. Tampering detection

Many of the previous security solutions hinder the speed of possible exploitation, but could be susceptible to brute-force attacks. For example, a simple password can be brute-forced to access the serial console, or link-time diversification of the devices can force an attacker to attempt exploitation multiple times before finding the correct offsets/password required for exploitation. The amount of tries for a successful brute-force attack can be too small for an effective deterrence due to technical constraints either in the device, such as small address space, or in the method used.

IoT devices often collect telemetry data of the device's performance or other aspect. We propose this telemetry should also include crash signaling. That is, if a device crashes due to a programming error, the next boot it signals about this alongside any other telemetry. This way one could collect the signs of attacks ongoing on the devices and possibly act upon this information to find and block the vulnerability being exploited. The telemetry signal would essentially be used as a measure of either a common crash being triggered across the device fleet or in worst case as a wide-scale alert system showing a possible vulnerability exploitation in progress by brute-force.

Other anti-tampering techniques also exist, such as chassis intrusion detection often found in power meters [34]. These can be used to strengthen the security by discarding encryption keys from memory in case of intrusion. They still require hardware-based support to function efficiently and are not always applicable.

Tampering detection can take various forms, but we score it at 3 as these changes can be very deep requiring the above mentioned telemetry data addition for example. This requires at minimum architectural changes. Code rewriting may also be required with USB as you need to modify the USB stack to account for validation data.

H. TLS integration

Cryptography is a necessity for almost any type of network communication to be considered secure. IoT devices are communicating with other devices and systems by definition. The most prevalent way to secure communications even in otherwise insecure networks is Transport Layer Security (TLS) [35]. We partially integrated mbed TLS library [36] into the Thingsee One as an attempt to strengthen its security. The mbed TLS is a TLS implementation library targeted towards embedded devices and should be one of the best candidates for TLS integration into less powerful IoT devices. During our integration experiment we ran multiple times into memory constraints and insufficient stack space, but we managed to slim down the functionality and size of the library enough to fit a mostly functional implementation into the device.

Performance of the resulting integration was over ten times slower than unencrypted communication, often exceeding 20 seconds for a single message. In addition, the amount of memory taken by the security library could easily be larger than the whole operating system itself, but this can often be tuned at the cost of either performance or functionality. The processor's acceleration features targeted towards encryption could have been used to boost performance, but we did not evaluate this. The performance boost would probably be significant, but it would still not reduce the memory usage, which was the biggest limiting factor based on our observations.

Based on the results, we can say that TLS with full protocol compatibility is very demanding for some IoT devices. The least powerful sensor devices can not hold in their limited memory all the security modes or certificates necessary to encrypt and decrypt the data being transmitted even when the devices are capable enough to perform HTTP and other messaging with the outside world. The devices would be limited to the very least a subset of a full TLS compatibility, which means we lose an important part of interoperability with web-servers and other parts of the network. This can be fixed by changing the hardware to a stronger one.

It can make sense to construct a more powerful central hub device for all the other IoT devices to connect to. The devices can then use a lightweight encryption between the hub and others parts of the internal network using protocols, such as what is defined in the IEEE 802.15.4 [37] for lightweight wireless communication. The hub can then implement full TLS for any external communications leaving resource constrained devices performing a specific efficient cryptography protocol. Unfortunately, adding a hub would clearly increase implementation complexity greatly, which means one should evaluate the cost between hub and more performant IoT devices already during the implementation of the device.

Similar to integrity verification, TLS integration is scored at 2 as it also requires some changes to architecture and may need hardware changes to accommodate the increased processing requirements. No code rewriting is required as TLS can often just be added as an extra layer that stops processing if validation fails. TLS is very widely supported, even in IoT, so no further research should be required.

TABLE I
COMPLEXITY-OF-IMPLEMENTATION SCORES OF THE PROPOSED SECURITY SOLUTIONS

Solution	C1	C2	C3	COI
Code obfuscation	0	0	0	1
Memory protection	0	1	0	2
Link time reordering	0	0	1	2
Stack shuffling	0	0	1	2
Integrity verification	0	1	0	2
TLS integration	1	0	0	2
Tampering detection	1	1	0	3
System call diversification	1	1	0	3

In Table I we summarize the discussed solutions with evaluated COI score and sort them by the score. With differing types of solutions, COI score presents diverse results, which indicates that this type of scoring could potentially be viable. Solutions which have a larger COI score compared to the median, tampering detection and the system call diversification, have existing implementations but require significantly more efforts to be implemented. Even though code obfuscation is scored below median, its nature of lowering performance should have taken into account when it is implemented.

V. CONCLUSION AND FUTURE WORK

In this study, we researched some of the application layer solutions for IoT security and discussed effective security measures for this environment. We reviewed the security measures that try to defend against the physical and application layer attacks. Although these solutions are mostly theoretical, we showed they could also be practical to implement. From all this, we analyzed the strengths and weaknesses of each solution, such as ease of use, possible system slowdowns and effectiveness against attacks. For each analyzed solution, we presented a COI score that evaluates the complexity of implementation.

Future work includes having practical test cases of the security solutions and through further analysis, discovering their effectiveness to strengthen the security in the context of IoT. In addition, we could not find any empirical surveys on the prevalence of various security solutions on IoT systems. This type of survey could provide insight for finding which solutions are a necessity for IoT or were easy to implement. A necessity means it may already have been used in an attack and is therefore an obvious solution to add. An easy solution may not be targeted for a specific attack, but is instead implemented as a proactive measure where the cost-benefit analysis of adding it makes sense. The survey could then improve the focus of any future security research and implementation efforts.

ACKNOWLEDGMENTS

The research leading to these results was supported partly by Tekes – the Finnish Funding Agency for Innovation, DIMECC Oy and Cyber Trust research program. This research is also supported by Tekes project 1881/31/2016 "Cybersecurity by Software Diversification". The authors gratefully acknowledge Henri Ståhl for researching security methods, especially for efforts on integrating TLS into an IoT device, Pertti Ranttila for researching alternative IoT devices for the security research and both for being part of the research team.

REFERENCES

[1] I. T. Union, "Overview of the Internet of things. Recommendation ITU-T Y.2060," 2012.

[2] M. E. Porter and J. E. Heppelmann, "How Smart, Connected Products Are Transforming Competition," https://hbr.org/2014/11/how-smart-connected-products-are-transforming-competition, accessed: 2016-06-25.

[3] R. Mahmoud *et al.*, "Internet of things (iot) security: Current status, challenges and prospective measures," in *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, Dec 2015, pp. 336–341.

[4] P. Koopman, "Embedded system security," *Computer*, vol. 37, no. 7, pp. 95–97, July 2004.

[5] H. J. Kim *et al.*, "A Study on Device Security in IoT Convergence," in *2016 International Conference on Industrial Engineering, Management Science and Application (ICIMSA)*, May 2016, pp. 1–4.

[6] O. Hahm *et al.*, "Operating systems for low-end devices in the internet of things: a survey," *IEEE Internet of Things Journal*, vol. PP, no. 99, pp. 1–1, 2015.

[7] P. Gaur and M. P. Tahiliani, "Operating Systems for IoT Devices: A Critical Survey," in *Region 10 Symposium (TENSYMP), 2015 IEEE*, May 2015, pp. 33–36.

[8] A. Riahi et al., "A systemic approach for iot security," in *2013 IEEE International Conference on Distributed Computing in Sensor Systems*, May 2013, pp. 351–355.

[9] J. H. Han, Y. Jeon, and J. Kim, "Security considerations for secure and trustworthy smart home system in the iot environment," in *Information and Communication Technology Convergence (ICTC), 2015 International Conference on*, Oct 2015, pp. 1116–1118.

[10] J. Singh et al., "Twenty security considerations for cloud-supported internet of things," *IEEE Internet of Things Journal*, vol. 3, no. 3, pp. 269–284, June 2016.

[11] I. Alqassem and D. Svetinovic, "A taxonomy of security and privacy requirements for the internet of things (iot)," in *2014 IEEE International Conference on Industrial Engineering and Engineering Management*, Dec 2014, pp. 1244–1248.

[12] J. Wurm *et al.*, "Security analysis on consumer and industrial iot devices," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2016, pp. 519–524.

[13] S. Hosseinzadeh *et al.*, "Security in the Internet of Things through Obfuscation and Diversification," in *Computing, Communication and Security (ICCCS), 2015 International Conference on*, 2015, pp. 1–5.

[14] C. Pohl and H.-J. Hof, "Secure scrum: Development of secure software with scrum," *arXiv preprint arXiv:1507.02992*, 2015.

[15] H. H. AlBreiki and Q. H. Mahmoud, "Evaluation of static analysis tools for software security," in *Innovations in Information Technology (INNOVATIONS), 2014 10th International Conference on*, Nov 2014, pp. 93–98.

[16] G. J. Holzmann, "The power of 10: rules for developing safety-critical code," *Computer*, vol. 39, no. 6, pp. 95–99, 2006.

[17] "Thingsee One," https://www.hackster.io/thingsee/products/thingsee-one, accessed: 2016-06-23.

[18] H. Shacham et al., "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*. ACM, 2004, pp. 298–307.

[19] ARM, "mbed uVisor," https://www.mbed.com/en/technologies/security/uvisor/, (Accessed on 09/23/2016).

[20] ——, "Cortex-M3 Processor," http://www.arm.com/products/processors/cortex-m/cortex-m3.php, accessed: 2016-7-29.

[21] Nuttx.org, "STM32 Null Pointer Detection," http://nuttx.org/doku.php?id=wiki:howtos:stm32-null-pointer, 2013, accessed: 2016-7-29.

[22] NuttX, "NuttX Protected Build," http://nuttx.org/doku.php?id=wiki:howtos:kernelbuild#the_memory_protection_unit, (Accessed on 09/30/2016).

[23] S. Hosseinzadeh *et al.*, "Security in the internet of things through obfuscation and diversification," in *Computing, Communication and Security (ICCCS), 2015 International Conference on*, Dec 2015, pp. 1–5.

[24] S. Hosseinzadeh, S. Hyrynsalmi, and V. Leppänen, *Obfuscation and Diversification for Securing the Internet of Things (IoT)*. Elsevier, 2016, p. 259–274.

[25] A. Homescu *et al.*, "Profile-guided automated software diversity," in *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, Feb 2013, pp. 1–11.

[26] T. Unangst, "Developing software in a hostile environment," 2014. [Online]. Available: https://www.openbsd.org/papers/dev-sw-hostile-env.html

[27] B. Martin et al., "2011 cwe/sans top 25 most dangerous software errors," *Common Weakness Enumeration*, vol. 7515, 2011.

[28] R. Buyya and A. V. Dastjerdi, *Internet of Things: Principles and Paradigms*. Elsevier, 2016.

[29] S. Rauti et al., "Towards a diversification framework for operating system protection," in *Proceedings of the 15th International Conference on Computer Systems and Technologies*. ACM, 2014, pp. 286–293.

[30] A.P. Syvertsen, "USB connectivity in a battery-powered IoT world," http://www.embedded.com/design/power-optimization/4439531/USB-connectivity-in-a-battery-powered-IoT-world, accessed: 2016-07-26.

[31] J. Larimer, "Beyond autorun: Exploiting vulnerabilities with removable storage," Blackhat, 2011.

[32] Microsoft Corporation, "USB FIlter," https://msdn.microsoft.com/en-us/library/dn638283(v=winembedded.82).aspx, accessed: 2016-07-28.

[33] Cisco, "Telnet, Console and AUX Port Passwords," http://www.cisco.com/c/en/us/support/docs/ios-nx-os-software/ios-software-releases-110/45843-configpasswords.html, (Accessed on 09/30/2016).

[34] E. Davis and B. Schafer, "Optical chassis intrusion detection with power on or off," May 14 2002, uS Patent 6,388,574. [Online]. Available: https://www.google.com/patents/US6388574

[35] R. E. Dierks T., "The Transport Layer Security (TLS) Protocol Version 1.2," https://tools.ietf.org/html/rfc5246, accessed: 2016-07-20.

[36] A. Holdings, "mbed TLS," https://tls.mbed.org/, accessed: 2016-07-20.

[37] N. Sastry and D. Wagner, "Security considerations for ieee 802.15. 4 networks," in *Proceedings of the 3rd ACM workshop on Wireless security*. ACM, 2004, pp. 32–42.