Journal of
Sensor and Actuator Networks

MDPI

Article

Internal Interface Diversification as a Security Measure in Sensor Networks †

Sampsa Rauti *, Lauri Koivunen, Petteri Mäki, Shohreh Hosseinzadeh, Samuel Laurén, Johannes Holvitie and Ville Leppänen

Department of Future Technologies, University of Turku, 20140 Turku, Finland; lauri@koivunen.work (L.K.); mapema@utu.fi (P.M.); shohos@utu.fi (S.H.); samuel.lauren@utu.fi (S.L.); jjholv@utu.fi (J.H.); ville.leppanen@utu.fi (V.L.)

* Correspondence: sampsa.rauti@utu.fi

† This paper is an extended version of Koivunen, L.; Rauti, S.; Leppänen, V. Applying Internal Interface Diversification to IoT Operating Systems. In Proceedings of the 2016 International Conference on Software Security and Assurance (ICSSA), St. Polten, Austria, 24–25 August 2016; pp. 1–5; Mäki, P.; Rauti, S.; Hosseinzadeh, S.; Koivunen, L.; Leppänen, V. Interface Diversification in IoT Operating Systems. In Proceedings of the 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC), Shanghai, China, 6–9 December 2016; pp. 304–309.

Received: 15 December 2017; Accepted: 2 March 2018; Published: 6 March 2018

Abstract: More actuator and sensor devices are connected to the Internet of Things (IoT) every day, and the network keeps growing, while software security of the devices is often incomplete. Sensor networks and the IoT in general currently cover a large number of devices with an identical internal interface structure. By diversifying the internal interfaces, the interfaces on each node of the network are made unique, and it is possible to break the software monoculture of easily exploitable identical systems. This paper proposes internal interface diversification as a security measure for sensor networks. We conduct a study on diversifiable internal interfaces in 20 IoT operating systems. We also present two proof-of-concept implementations and perform experiments to gauge the feasibility in the IoT environment. Internal interface diversification has practical limitations, and not all IoT operating systems have that many diversifiable interfaces. However, because of low resource requirements, compatibility with other security measures and wide applicability to several interfaces, we believe internal interface diversification is a promising and effective approach for securing nodes in sensor networks.

Keywords: interface diversification; security; software; Internet of Things

1. Introduction

The Internet of Things (IoT) is a network composed of physical devices and other objects [1]. These "things" are incorporated with software, electronics, sensors and actuators. By using network connectivity, the devices collect and exchange information with each other. Today, IoT devices are used by various public and private sectors from health care to industrial applications. The whole physical infrastructure has become closely connected by information technology with the help of sensors, as interconnected embedded devices transmit measurement results and instructions over the network. The IoT forms a global infrastructure and covers several applications areas, such as smart cities, smart grids and power plants, with the aim of making everyday life more effortless and comfortable. In a wireless sensor network (WSN), several nodes are equipped with sensors that detect physical or environmental conditions, such as the temperature, light or pressure. With the rapid development of sensors, WSNs can be considered a key technology in the IoT environment.

More and more new devices and sensors are connected to the Internet daily, and the IoT keeps growing. Gartner estimates over 6 billion "things" are in use now, and there will be about 20 billion by 2020. Another prediction is that more than half of major new business processes and systems will incorporate an IoT component by 2020 [2]. The enormous growth in the number of IoT devices also means there are serious implications for software security. The International Telecommunication Union (ITU) states in its recommendation that the "IoT makes full use of things to offer services to all kinds of applications, whilst ensuring that security and privacy requirements are fulfilled." [1]. In practice, however, security and privacy are still a major challenge for the burgeoning IoT environment. The mismatch between the growing significance and the poor security of the IoT is obvious. HP reported that at least 70 percent of IoT devices are vulnerable to exploits that may be harmful to users [3]. Not surprisingly, adversaries have already started taking advantage of this issue, as was clearly shown with Mirai, a piece of malware found in August 2016. Mirai launched large-scale network attacks by using a botnet comprising of hundreds of thousands of IoT devices [4].

There is a clear incentive to improve software security in the IoT environment. Cybersecurity vendors and service providers have noticed the problem and are already devising new approaches and architectures for IoT security. However, novel security solutions that can effectively protect devices while taking into account the very limited resources available in the IoT environment, as well as sensor and actuator networks, are still needed. In our view, interface diversification is a security scheme that shows a lot of promise in the IoT environment.

In the current era of *software monoculture*, not many execution platforms exist with large differences. Instead, there are huge numbers of copies of the same execution platform with the same internal structure. Because of this, one piece of malware can potentially infect millions of devices. Malware uses knowledge about the system's internal interfaces to achieve the attacker's objectives. If the internal interfaces were different in each IoT device, malware that relies on the information about the well-known identical interfaces would not be able to run in a system [5,6]. The idea of *internal interface diversification* follows this line of reasoning. Unique diversification applied to the target node's internal interfaces makes its software much more resistant to potential cyber attacks. In a system that has been diversified, a malicious piece of software cannot communicate in the secret "language" used in that system and, therefore, is prevented from accessing the the system's resources.

In this paper, we propose interface diversification as a security measure for devices in sensor networks. The contributions of this paper are as follows. First, based on our previous conference paper [7], we present a study on diversifiable interfaces in IoT operating systems. This study has been extended here considerably with a discussion of 10 additional operating systems. Second, we offer two proof-of-concept implementations for the IoT environment that demonstrate the feasibility of diversification. These solutions were first introduced in [8]. In this paper, we provide more implementation details and an extended discussion of these solutions in the context of sensor networks. Based on our experiments, we evaluate the feasibility, strengths and drawbacks of interface diversification in the IoT environment. We also discuss attack scenarios that are mitigated by interface diversification in sensor networks.

The rest of the paper is structured as follows. Section 2 introduces interface diversification and related concepts. We also introduce some common attack scenarios in sensor networks that we aim to combat with interface diversification. Section 3 reviews related work in the field of diversification research. Section 4 explains how the operating system interface diversification can be applied to IoT operating systems and presents a review of diversifiable interfaces in 20 commonly used IoT operating systems. Section 5 presents our proof-of-concept implementations for two IoT operating systems. We provide practical implementations for memory layout shuffling in Thingsee OS and symbol diversification in Raspbian operating systems. Furthermore, experiments with these solutions are conducted to show the feasibility of interface diversification in the IoT environment. Section 6 discusses interface diversification as a security measure in sensor networks and reviews its strengths and weaknesses. Finally, Section 7 concludes the paper.

2. Interface Diversification

2.1. The Idea of Interface Diversification

By diversifying interfaces, we modify applications' or systems' internal interfaces with the aim of making them unpredictable for a malicious adversary. In other words, diversification severely limits the number of assumptions an adversary can make about the surrounding system where the malicious code is meant to be run. In this paper, we concentrate on the diversification of internal interfaces; that is, we do not diversify the external interfaces exposed to the end-user. In practice, diversification can be implemented by using several code obfuscation techniques [5]. Even very simple obfuscation techniques, such as renaming identifiers and changing the order of parameters in function signatures, can be used to diversify the internal interfaces.

When we use the term *interface*, we refer not only to ordinary interfaces provided by libraries and software modules. We use the term broadly to cover many other things in the system that can be seen as interfaces and benefit from diversification. Examples include commands of a scripting language or memory addresses (of critical functions or services). When understood in this manner, an interface is anything that the adversary can potentially use to gain access to critical services or resources in the system.

As a practical example of interface diversification, consider altering the mapping of system call numbers in an operating system. Applications use system calls to use the resources of a computer and to request a service from an operating system. Basically, in order to accomplish anything meaningful, a malicious program has to use system calls to operate. Therefore, it makes sense to diversify the system calls to keep the adversary from exploiting them.

In the Linux operating system, for example, system call diversification can be implemented by replacing the original system call numbers—more than 300 system calls defined in the system call table—with new ones. Because system calls are used by many library functions (wrappers), diversification also has to be propagated to the code that uses the system calls. In other words, the system call numbers are updated to their diversified counterparts in all libraries and applications that invoke system calls directly. In addition, the library functions that invoke systems calls directly or indirectly are renamed so that the malicious software cannot use these functions. Put differently, the *transitive closure* of the system calls has to be diversified so that the malware cannot exploit these entry points to the critical resources of the computer.

Diversification of system call numbers is performed uniquely for each operating system installation, meaning that each system or device gets unique protection. This way, the adversary cannot use the information gained by compromising a single system in a widespread attack targeting other systems. In the Linux operating system, for example, the system calls are represented by 64-bit numbers; therefore, a brute force attack to guess the secret diversified system call numbers is laborious and can be easily detected.

2.2. Attack Scenarios Mitigated by Diversification

Generally speaking, interface diversification mitigates (1) attacks where the adversary manages to bring a harmful executable file in the system and tries to run it and (2) attacks where malicious instructions are slipped in an existing process with the aim of executing these instructions in the memory space of that process. We present several attack scenarios that are prevented by diversification in the context of sensory networks.

2.2.1. Infecting the Node

Internal interface diversification protects against scenarios where the adversary has succeeded in slipping a harmful executable into the system and attempts to run it. The aim of the attacker is to get full control over the IoT device, which can then be used according to his or her objectives. However, if the malicious executable fails to successfully invoke the diversified system calls (or run commands

on a diversified command shell), the malicious executable cannot operate in the system. Regardless of whether the attacker has gained physical access to the IoT device or is operating remotely from another machine in the network, interface diversification makes running malicious executables in the system considerably more difficult. Therefore, although internal interface diversification does nothing to stop malicious programs from being planted in the system, diversification prevents them from running. This also applies to many previously unknown pieces of malware that would go unnoticed by anti-virus software. In this respect, internal interface diversification is a proactive measure.

2.2.2. Worm Propagation

Another potential attack scenario in the IoT environment is self-replicating and spreading malware. An example of such a worm is Mirai. Mirai turned infected devices into a botnet that can be exploited in large-scale network attacks. It scans devices for telnet, uses a hard-coded list of common credentials to access devices and starts running malicious code [4]. In the current software monoculture, this kind of worm is potentially compatible with millions of nodes, which allows it to spread uncontrollably. However, the malicious code cannot run on a diversified system even if the worm gets past the authentication. Consequently, diversification also mitigates the propagation of malicious software.

2.2.3. An Infected Update

Internal interface diversification can be used to mitigate the threat of infected updates. The adversary can try to slip malicious code into a remote update at some point along the way. Of course, the remote updates are usually sent in encrypted form and possibly signed, but the keys might get compromised, or the adversary could find a way to tamper with the contents of the update before encryption. Paradoxically, this causes the updates often meant to improve the security of the system to become a potential security risk themselves. However, as long as the adversary does not know the diversification secret of a specific node or insert malicious code before the update is diversified, the harmful code slipped into the update does not work in a diversified system. Of course, if the attacker gets access to the diversified code, he or she can attempt to deduce its meaning and find out the results of the diversification that way. However, this method requires costly analysis, and the results can be used only in that one specific system.

Of course, having diversification in place can make distributing updates to systems more challenging as the updates need to be compatible with the diversified system. This ties into the question of when the diversification should be performed. If the updates are received pre-diversified, the vendor delivering the updates needs to be aware of the diversification secret. If the updates are diversified on-device before they are applied, the system must pose a way to diversify, and subsequently run, undiversified binaries. Securing these pathways is critical to the security of the entire scheme: if an attacker can find a way to diversify untrusted binaries with the device-specific diversification secret, then he or she is able to run arbitrary code.

2.2.4. Compromising Node's Integrity

Attackers may also aim to attack a sensor in order to change the data that has been collected or to compromise the integrity of the sensor's software. Internal interface diversification can be used to supplement the security measures related to node integrity. For example, remote attestation [9] is a mechanism usually used to measure the internal state of a sensor node. The resulting information is reported to a remote verifier so that the node's integrity can be checked. However, the drawback of the method is that it concentrates on executable files during the loading phase but usually does not prevent runtime attacks. Therefore, extra protection for runtime attacks, such as those that exploit buffer overflow vulnerabilities, is needed. In a buffer overflow, a program accidentally overruns the buffer's boundaries and overwrites the data in memory. This vulnerability can be exploited by writing malicious code into areas where executable code is held.

In a diversified system, however, the harmful code would use the wrong (undiversified) functions or system calls and as a result could not run successfully. Verification of the integrity of the static boot time system would not be able to catch attacks of the kind that exploit vulnerabilities in trusted pre-approved binaries. In contrast, diversification can protect against attacks that work by introducing new code into a vulnerable process. The attacker would still be able to inject code, but the code would likely not function within the context of the diversified system. In this way, we view diversification as an important protection mechanism to be used in addition to other forms of defense. Internal interface diversification can also act as a replacement for other runtime protection mechanisms, such as data execution prevention (DEP), that have not necessarily been implemented in many IoT operating systems.

3. Related Work

In the literature, there is a large body of research on securing operating systems with diversification, which is applicable to IoT and sensor networks. Diversification is a promising approach in securing software and applications and has benefited many different environments. There exist several diversification methods for thwarting various types of security attacks [10,11]. The methods consider various interfaces as the targets of diversification, such as system calls, functions in shared libraries, command shells, memory space and protocols.

Hosseinzadeh et al. [12,13] propose diversification of operating system interfaces of IoT devices as a way of presenting security for these networks at the application layer. This method is promising for protecting largely distributed networks and mitigating massive-scale attacks. The authors also discuss diversifying the communication protocols used among the devices of these networks for presenting security at the network layer.

A communication protocol can be regarded as an interface, and, therefore, as a potential target for obfuscation with the aim of protecting the confidentiality and integrity of the communication. Protocol obfuscation makes it more difficult for an intruder to identify the protocol through static analysis and recognize the protocol using traffic classification tools. The obfuscation could be done by removing the identifiable properties of the protocol, such as packet size and byte sequence, in such ways that they look random. This kind of protocol obfuscation closely resembles cryptography as a security measure [14].

Chew and Song [15] propose three lightweight randomization techniques for operating systems that increase the heterogeneity of the system and impede buffer overflow exploits. These techniques rely on randomizing the global library entry points, system call mappings and stack placement. Typically, library exploits start with gaining knowledge about the address of the library routines and exact address of the entry point of the function. In the scenario that all the shared libraries are mapped in the same virtual address, it is very likely that the attacker could guess the function's entry points. Here, one way to change the function's entry point is to change the function to a dummy function with the same functionality and change the name of the original function. A malicious program that does not know the new name and entry point will not succeed in using the function. In this way, the adversary is prevented from gaining access to root shell by using functions in shared libraries.

System calls are a programmatic way for applications to request services from the OS kernel. If not well protected, system calls could also be used by malware to access system resources. Diversifying system calls is an effective way of impeding such a threat. Randomizing system call mappings [15] is a method for avoiding malware from directly making system calls. More specifically, changing the mapping of the system calls and the correlated numbers in the system call table decreases the probability that the malicious code can invoke a correct system call.

System call diversification has also been studied by Rauti et al. [16,17] as a means of protecting operating systems from malware. This can be done by altering system call numbers in the operating system's kernel and accordingly altering the applications invoking these system calls [17]. This can be implemented by diversifying system calls in ELF (Executable and Linkable Format) binaries [16].

Ergo, the malware that is not aware of the new system call numbers is unable to interact with the environment. Symbol diversification [18,19] diversifies the symbol names used with dynamic linking of shared libraries. Shared libraries implement functions that are used by other libraries or executables. In order to find these functions, symbol names are associated with them. Diversifying the symbol names makes the functions unknown to malware that does not have knowledge about the diversification secret.

Diversifying shell commands is a technique for protecting the system from malware. Uitto et al. [20] present a diversification approach that prevents attacks (e.g., injection type of attacks) in the command shell environment, through diversifying Linux shell commands. After these shell commands are diversified, a piece of malware no longer has knowledge about the correct commands and therefore, cannot use the shell to pursue the malware's goals.

In another work, Uitto et al. [21] introduced an instruction set diversification scheme for interpreted languages to restrain the execution of malicious scripts. In this diversification method, programming language interfaces are diversified or renamed in such a way that they appear unknown to adversaries, and malware is not able to attach itself to the system. The diversified scripts generated by this diversification tool can be run only by an interpreter that supports these diversified scripts, using the diversification secret. This means that malicious adversaries that do not have the diversification secret cannot diversify their code correctly, and their execution will be prevented.

Diversification of memory space has been widely studied as a means of obscuring the related memory addresses, by randomly changing the start address of the executable and the positions of the heap, stack, and libraries. Thus, it is difficult for an attacker to predict the target addresses. Address space layout randomization (ASLR) is an effective randomization memory-protection technique for thwarting memory corruption, code reuse, code injection, and buffer-overflow attacks. Shacham et al. [22] demonstrate the effectiveness of address space randomization in breaking the software monoculture and hindering an attacker from exploiting different randomized instances of a program, utilizing the same attack code. They also show that re-randomizations, after the initial address space randomization can amend security in resisting brute force attacks.

Because the stacks of all the processes start at the same address in Linux2.2.x, it is simple for a malicious shell code to guess the correct return address beforehand. In this regard, Chew and Song [15] propose randomizing the start address for the stack of each process, that is, allowing stacks to start from random positions, in order to make the stack-based exploits more challenging.

The implementation proposed by Bhatkar et al. [23] focuses on address obfuscation, by randomizing the base address of the heap, stack and code segments and adding random padding to *malloc* function calls and the stack frame. This binary rewriting tool rewrites the object files and executables to randomize the addresses. It also periodically re-obfuscates the executables and libraries, in order to prevent an attacker from probing the fixed address space layout.

Höller et al. [24] used ASLR as an effective technique for not only preventing malicious attacks but also for increasing the fault detection rate of memory-related software bugs. The authors propose an adaptive dynamic software diversity method that designs the program in such a way that during execution, it could be randomized based on the parameters that are adjusted at runtime. More specifically, their approach randomizes the base addresses of the stack, the shared memory regions and data segment, and the virtual dynamic shared object.

Although a number of studies proposed diversification of different interfaces, this research area is still very new in the context of the IoT and sensor networks. Our work in this paper adds to the current body of knowledge on IoT security by proposing diversification for securing sensor networks and providing practical proof-of-concept implementations demonstrating the feasibility of this approach. Further study of the applicability of interface diversification techniques in the IoT environment is an important topic for future research.

4. Diversifiable Interfaces in IoT Operating Systems

To get a better understanding of the suitability and feasibility of diversification as a security measure in IoT operating systems, we conducted a study on diversifiable interfaces found in common IoT operating systems. Several interfaces have been suggested as potential targets for diversification in the existing literature, as we saw in the previous section. Based on the existing literature and our research, we chose the following interfaces as the study targets:

1. *System calls.* A system call is the way programs request a service from the operating system and access the computer's critical resources [25]. To prevent the attacker from using system calls, we can uniquely change the system call numbers [16,26,27]. This interface diversification is then propagated to trusted binaries and libraries, which are diversified accordingly so that they are compatible with the diversified system and can invoke system calls using the new secret system call numbers.
2. *Functions in shared libraries.* System calls can also be invoked indirectly by calling several wrapper functions in many operating system libraries. Malware should be prevented from accessing any critical resources by diversifying these entry points. To this end, any library functions that directly or indirectly issue system calls are diversified [15,18]. This means changing function names and possibly changing the order of parameters in the function signature. Diversification is also propagated to libraries and applications that call these library functions.
3. *Command shell.* Malware does not always use system calls or library functions. Instead, it can employ interpreted languages, such as shell scripts, to achieve its goals. Similarly to the library functions, the language interface of the shell is also an entry point to the resources of a device. Many attacks, such as ShellShock [28], have seized this opportunity. If we change the language interface (the set of tokens used by the command line interpreter), the attacks based on this known language interface will fail [20,21,29]. The shell scripts in the system are then diversified to correspond to the new secret language.
4. *Memory space.* Memory space can also be seen as a diversifiable interface. ASLR is a security approach that provides protection from buffer overflow attacks. In ASLR, the address space positions of the essential parts of a process are randomly rearranged in order to make it more difficult for the adversary to predict where a specific piece of code resides in the memory. With the memory layout randomized, the attacker cannot reliably jump to a specific position in the memory (such as an exploited function) [30]. If one wants to further increase the chance of an attacker guessing the location of a specific randomly placed memory area, it is possible to further increase the amount of virtual memory space where the diversification occurs. To achieve this, however, a memory management unit (MMU) is needed. An MMU is a unit through which the memory references are passed. It takes care of translating the virtual memory addresses into physical addresses. A system without an MMU is limited to the address range of its physical space when ASLR is applied, whereas the MMU usually expands the address space, which allows the placement of mapped sections with much more potential offsets than a physical address space alone would permit.
5. *Data structures.* Related to memory layout randomization, the layout of individual data structures can also be diversified. For example, the order of data items within structures could be changed or additional padding inserted between them. This form of protection has been implemented in the main line Linux kernel: when the kernel is compiled, a special compiler plugin can be invoked to change the layout of the data structures used within the kernel [31]. Since these data structures are internal to the kernel, the changes do not necessarily need to be propagated to other parts of the system. The only challenge with this approach is compatibility with out-of-tree modules that need to be built separately.
6. *Protocols.* As IoT devices operate in a network, they need to use protocols to communicate. These protocols can also be seen as targets for diversification [14]. For example, the Constrained

Application Protocol (CoAP) is a software protocol that enables simple electronics devices to communicate with each other [32]. It is an application layer protocol specifically designed for resource-constrained devices, such as "things" in the IoT. By diversifying a protocol, we can create numerous uniquely diversified protocols from the original protocol. Thinking of a protocol as a state machine, this means that diversification can add arbitrarily many new states and transitions to the original protocol. Of course, this protection comes at the price of greater complexity and some performance slowdown. Protocol diversification makes node capture attacks and adding malicious nodes to a network more difficult, as those nodes also have to conform to the diversified protocol in order to successfully interact with other nodes.

The first five diversification schemes above have been suggested for traditional operating systems whereas the sixth is IoT specific. In the context of the IoT, circumstances may be somewhat different for these schemes. For example, some IoT operating systems do not provide an implementation of a system call interface, or this interface is not widely used by applications. In addition, not all IoT operating systems include a memory management unit. In the following sections, we will discuss these interfaces in the context of the IoT.

Diversifiable Interfaces in IoT Operating Systems

We studied the previously discussed interfaces in 20 IoT operating systems. Our findings are listed in Table 1. We included many notable IoT-friendly operating systems in the study, some commercial and many open-source. Our list of IoT operating systems certainly is not exhaustive and new ones are in the making as new use cases for the IoT are found. Still, we believe our survey gives a good picture of the current interfaces available in IoT operating systems.

Table 1. The diversifiable interfaces in various IoT operating systems.

OS	Source Code	MMU Support	System Calls	Shell
Android Things	Available	Yes	Yes [33]	Yes
ChibiOS	Available	No	Minimal [34]	Yes
Contiki	Available	No	No [35]	Yes [36]
Amazon FreeRTOS	Available	No	No [37]	Yes
GNU/Linux	Available	Required	Yes	Yes
Integrity	Closed	Yes	Yes [38]	No
MANTIS	Available	No	No [39]	Yes
mbed	Available	No/uVisor	Partial [40]	Yes
Mynewt	Available	No [41]	No [42]	Yes
Nano-RK	Available	No	No [43]	No
Neutrino	Closed	Yes [44]	Yes [45]	Yes
NuttX	Available	Partial	Yes [46]	Yes
Particle	Available	No	No [47]	Yes
RIOT	Available	No	No [48]	Yes
TinyOS	Available	No	No [49]	Yes
UIUC LiteOS	Available	No	Yes [50]	No
VxWorks	Closed	Yes	Yes [51]	Yes
Windows IoT	Closed	Yes	Yes [52]	Yes
Yocto Project	Available	Based on GNU/Linux		
Zephyr	Available	Yes [53]	Yes [54]	Yes

Table 1 shows the availability of the MMU support (not strictly necessary for memory layout diversification), system call interface and shell for each studied operating system. The availability

of source code is also listed as open-source systems are generally better suited for diversification, although diversification techniques can and often are also used on the binary level [55].

In some cases, for example, the cases of RIOT and TinyOS, the number of interfaces to which we can apply diversification appears to be limited without concrete use cases. However, many operating systems, like Linux and Android Things, contain several essential interfaces that would benefit from diversification and are relevant to protect. In addition, all the proprietary systems we studied seem to contain many diversifiable interfaces. However, actually using diversification on them may require close collaboration with the commercial entity due to inflexible default licenses. The lack of access to the whole source code of the system is also a challenge, as the diversification secret needs to be propagated throughout the whole system.

Android Things (formerly called Brillo), an operating system geared toward the IoT by Google, is still in developer preview, but from a security and diversification perspective, the architecture looks promising. This is achieved by requiring hardware that is richer in features [56] compared to most other devices used in the IoT domain. As MMUs have been scarce in the IoT environment, many other IoT operating systems lack the extra protection afforded by potentially compatible devices that would support an MMU.

In contrast to Android Things, the open-source operating system *Contiki* requires little memory but also has no support for a memory management unit or system call interface. Diversifiable interfaces are challenging to study on an operating system as small as Contiki as there is not a lot more within the system than the kernel itself. Contiki does include a shell where the language of the shell could be diversified, but the shell is not required by any part of a normal system and therefore, is optional. Another potentially diversifiable feature in Contiki is the networking protocol 6loWPAN [57], where the protocol could be regarded as an interface. Although 6loWPAN can be used with encryption [58], diversification can be used in combination with encryption to provide additional security.

ARM's *mbed* operating system can include a supervisor to isolate system components from each other without a full MMU in the hardware [40] but instead with a lightweight memory protection unit (MPU) support found in many ARM processors. This isolation is achieved using supervisor calls that are mostly analogous to system calls and therefore, are an interesting subject of study concerning use of diversification as an additional security reinforcement layer. The supervisor allows different isolated components to communicate with each other, which can be regarded as an internal interface that then could be attacked and exploited. This interface could be diversified in order to make it impossible for injected code to even be able to reach the supervisor functions.

The proprietary operating systems listed—*Integrity*, *Neutrino*, *VxWorks* and *Windows IoT*—claim to support the MMU and appear to have a possibility for system calls. These calls could probably be diversified for similar additional security as in ARM's mbed depending on how they are implemented. Licensing may potentially limit the diversification of commercial systems due to restrictions on reverse engineering and modifications, but that topic is beyond the scope of this paper.

Amazon FreeRTOS is a Real Time Operating System (RTOS) with a very minimal kernel but great extensibility. Although MMU support does not exist so the OS can be run on the supported hardware without an emulation, extensive MPU support similar to ARM's mbed [59] does exist. Essentially, the kernel and device peripheral access can be protected from memory reads and writes, which means there has to be an interface to talk to the kernel, which could then be diversified to make it more difficult for malicious code to interface with the kernel. FreeRTOS itself is relatively minimal and does not have much to diversify, but when actual functionality beyond core kernel functions is added, diversification becomes much more viable. The technique used in our proof-of-concept implementation on the Thingsee NuttX OS should also be applicable.

ChibiOS appears to have a very specific use case for system calls. It has only seven system calls within its source code [34], which is not enough to apply any effective interface-level diversification. ChibiOS does not appear to use any memory protection functionality, which makes the system insecure. Combining layout diversification and even minimal system call diversification could be beneficial.

The kernel may not be protected, but its location is not known, which forces the attacker to use a brute force approach on each device. Diversifying system calls in this case further protects the whole system as now the attacker also has to use brute force to be able to use the system calls to interface with the kernel.

Zephyr is an operating system developed by the Linux Foundation, which runs on multiple platforms ranging from x86 to ARM. Zephyr has a few similarities to Linux but does not necessarily require an MMU. An MMU is used only for the limited purpose of MPU functionality as used in mbed and FreeRTOS to achieve similar functionality as MPU-supported devices. As the operating system supports MPU-based kernel isolation, Zephyr has implemented a system call interface, which should be diversifiable.

Most of the studied operating systems include some type of command shell. Several real-world vulnerabilities, such as Imagemagick (CVE-2016-3714 [60]), are based on unescaped input, which makes it appealing to study the possibility of diversifying the shell language. However, many IoT operating systems currently do not expose the shell for use to the rest of the system, and the command shell is mainly geared toward debugging.

Although not covered in this study in detail, on a more fine-grained level, there are some operating system-specific APIs (Application User Interfaces) that could be diversified. For example, Contiki has a device driver API and a file system API. As these operating systems mature in the future, most will probably be supplied with an increasing number of diversifiable interfaces, such as shared libraries and operating system APIs, aimed to make building applications easier. Of course, the interfaces and libraries available for diversification also strongly depend on the application area and specific use case of a particular device.

Finally, even though all the software interfaces of the IoT operating systems we studied are definitely not perfect for diversification, they all support some diversifiable protocols. For example, Contiki offers a full IP stack which includes standard protocols, such as UDP, TCP and HTTP [36]. In addition, most IoT operating systems support new low-power protocol standards, such as CoAP (Constrained Application Protocol) and RPL (the IPv6 Routing Protocol for Low-Power and Lossy Networks). Although we will not go into the details here, all of these protocols can also be seen as targets for diversification.

To summarize, although many interfaces are still lacking in IoT operating systems, most systems expose potential targets for diversification of internal interfaces. Based on our findings here, we believe that IoT devices are ready for security improvements introduced by diversifying the internal interfaces.

5. The Proof-of-Concept Implementations

In this section, we introduce two practical proof-of-concept implementations for internal interface diversification in IoT operating systems to gauge the feasibility of interface diversification in the IoT environment. The description in this section is deliberately technical and practical to give the reader a good understanding of how these tools work and to make our experiments reproducible. We constructed a modified linker for diversifying the memory layout of Thingsee OS and demonstrated how this can prevent an exploit attempt from working on a Thingsee One device (Section 5.1). Additionally, we applied symbol diversification to executable files on Raspbian OS and performed experiments with this implementation (Section 5.2). The goal was to apply the interface diversification to existing systems and test our implementations against a working exploit to evaluate the feasibility and effectiveness when thwarting malicious attacks.

5.1. Layout Shuffling of Thingsee OS on Thingsee One

Many resource-constrained and often time-critical IoT devices and operating systems do not support many ordinary security measures, such as virtual memory and no-execute bits, that are commonly found in modern desktop and server computing platforms [61]. This is also the case with

the default "flat" NuttX-based Thingsee OS build. If there is no memory protection, specialized system call traps, or dynamic shared libraries, the prospects for different diversification schemes are limited.

We decided to apply memory layout diversification, previously studied by [62], to Thingsee OS with a flat address space configuration. The idea in diversifying the memory layout is to prevent attacks based on certain data or code residing at known addresses from functioning. One such example is return-oriented programming (ROP) attacks [63,64], where the attacker hijacks the program control flow and causes the execution to jump to carefully chosen machine instruction sequences already stored in the device's memory.

The implementation of the diversification scheme required introducing memory layout shuffling to the GNU linker. Our modified version of the GNU ld linker was created with just a few modifications to the original source code. To obtain the desired functionality, three files had to be edited, with about 100 added and a few deleted or modified source code lines. GNU ld already contains the functionality to sort sections whose names match a wildcard expression defined in a linker script by name or alignment. We introduced an additional sorting method that sorts the wildcard-matching sections by salted md5 hashes of their names. There should be no need to use the same salt more than once. This is because the memory layout itself is an interface that is not needed by legal code, other than the linked program itself which has the appropriate addresses provided by the linker. Our diversification scheme could still be optimized in terms of performance and memory consumption, but it worked without problems in our experiments. To increase diversity in the linked software, the source code should be compiled to object files in a way that all data objects and functions get their own section. With *gcc*, this means passing *ffunction-sections* and *fdata-sections* as parameters for the compiler.

We then wrote a vulnerable Thingsee OS application program that writes user-supplied data to a stack buffer without bounds-checking. We then devised an exploit that overwrites a return address on the stack and makes the program jump to a chosen function. After executing the function, the system apparently crashes and reboots, preventing the execution of the malicious code. Later, we rebuilt Thingsee OS using our modified *ld* with randomization of the order of functions and data in the Thingsee OS image. An identical exploit did not cause the chosen evil function to be executed this time. Instead, it made the program crash immediately, as intended.

Our vulnerable program was created simply by modifying the "Hello World" example in the Thingsee OS in order to facilitate creating a new Thingsee OS application. Our vulnerable program takes an input from a command line argument as a hexadecimal string and writes it to a stack buffer in a hex-decoded form without bounds-checking. We use this input method to simulate a malicious input that could be given in a real application environment, such as a network server.

We compiled Thingsee OS on Lubuntu 16.04 x86_64 using arm-none-eabi-gcc version 4.9.3 and our modified versions of binutils-gdb (from git commit *6e2565079204ae2d2c0a5fa15fcd233e9c614f0b*), and thingsee-sdk (from git commit *b65cfa8ec466d498e24959b90568b076c942aa6d*). A forked repository for Thingsee can be found at [65] with our modifications. The fork of binutils-gdb can be found at [66] with a layout shuffling implementation.

We built the undiversified version of Thingsee OS. After building, we looked up the address of the evil function for exploitation from the firmware with object file disassembler, which shows us the addresses of all functions within the firmware:

```
$ arm—none—eabi—objdump —d nuttx
```

The hello-function was found at address 0x08014174 and the address for evil function at 0x08013fcc. These offsets are required for successful code execution exploitation.

The following listing shows the Thingsee OS command line shell. First, we examined the stack to find where the return address resides inside the hello-program (**b5410108**). We then devised and ran the exploit. The ARM processor needs the return address be incremented by one when using Thumb code [67]. Thus, we add 1 to the address of 'evil' (08013fcc) and write it to the stack in little-endian byte order (**cd3f0108**).

```
NuttShell (NSH)
nsh> hello 42 0 20
hello 42 0 20
hex_strcpy: a=20009030; s_d=0; s_n=32
00000000000000000000000000000064900020b5410108754101080000000
foo 1
B
nsh> hello 101122334455667788 99aabbccddeeff11111111cd3f0108
hello 101122334455667788 99aabbccddeeff11111111cd3f0108
hex_strcpy: a=20009010; s_d=0; s_n=32
00000000000000000000000000000044900020b5410108754101080000000
foo 1
(some unintelligible characters)
Evil function!
```

After printing "Evil function!", the USB connection to our device breaks as the system crashes.

The layout-shuffled version of Thingsee OS was built as follows:

```
cd configs/haltian-tsone/
patch Make.defs <<END_OF_PATCH
79,80c79
< LDFLAGS += --gc-sections
< # --sort-section=shuffle_obfuscation
---
> LDFLAGS += --gc-sections --sort-section=shuffle_obfuscation
END_OF_PATCH
cd ../../tools/
./configure.sh haltian-tsone/nsh
make menuconfig # Enable the ''Hello, World!'' example
env SHUFFLE_OBFUSCATION_SALT=foo make

# NOTE: set the device to flash mode before executing,
# the hexadecimal number specifies the device
sudo dfu-util -d 0483:df11 -a0 -D nuttx.dfu -s :leave
```

First, we patch the Thingsee configuration to enable our implementation of linker diversification. For build reproducibility, randomness for the diversification can be provided by supplying the build process with the environment parameter *SHUFFLE_OBFUSCATION_SALT*. This parameter would be varied for each diversified device build. Finally, we flash the diversified firmware into the device to attempt exploitation again.

The same exploit was run again:

```
NuttShell (NSH)
nsh> hello 101122334455667788 99aabbccddeeff11111111cd3f0108
hello 101122334455667788 99aabbccddeeff11111111cd3f0108
hex_strcpy: a=20009030; s_d=0; s_n=0

foo 1
(some unintelligible characters)
```

Again, the connection is broken. However, now the function 'evil' is not called. Undefined behavior is invoked nevertheless, and Thingsee OS crashes. In this particular experiment, execution happens to jump to halfway in the middle of a 4-byte instruction word that is part of a tiny piece of code that jumps to an exception handler. Figure 1 shows the execution in both cases described above.

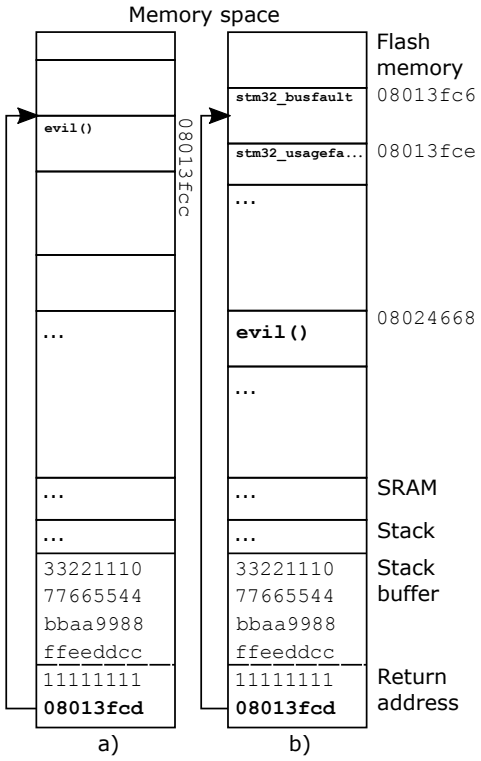

Figure 1. An example of a possible execution of the exploit in a normal and diversified operating system. (**a**) normal OS: execution jumps to the beginning of the function evil; (**b**) OS layout diversification: in this particular case, the execution jumps to the middle of an instruction.

A crude estimate of the granularity of Thingsee OS layout shuffling can be seen in Figure 2. It shows a histogram representing the distribution of bytes in differently sized *PROGBITS* sections in the source object files of Thingsee OS. We can see that 6.2 MB out of total 20 MB is found in sections at most 1000 bytes in size.

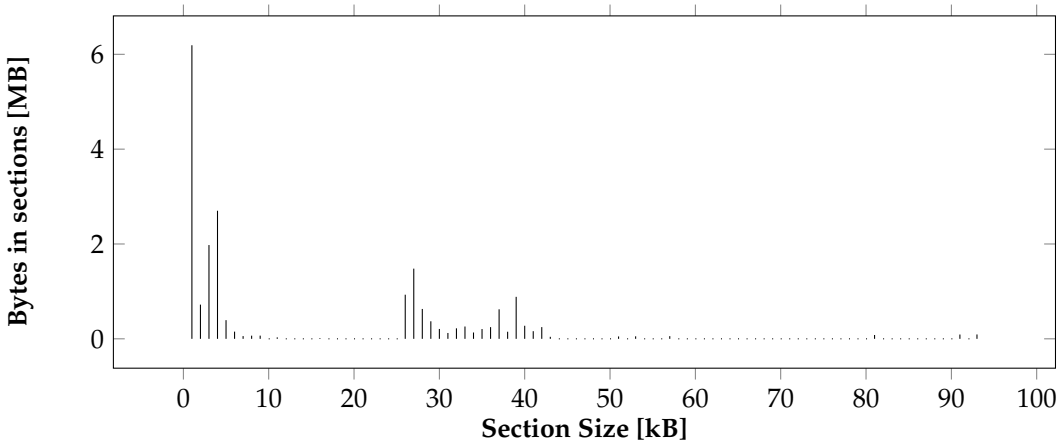

Figure 2. A histogram of the distribution of bytes in differently sized *PROGBITS* sections in the source object files of Thingsee OS.

We measured the diversified and undiversified binary sizes of the resulting firmware. The default firmware configuration for Thingsee with our toolchain weights 6002288 bytes or around 5.8 megabytes. We measured the shuffled binary size five times with different shuffling seeds. The binary grew at minimum 200 bytes and maximum of 272 bytes. On average, the size increased by 0.004%. The growth

is explained by alignment constraints due to hardware requirements. The linker may push a block's offset towards the end of the file to achieve the required alignment for each block. On a device with no alignment constraints, diversification does not increase the file size at all.

5.2. Symbol Diversification on Raspbian

The symbol diversifier tool renames the symbols in shared libraries. In addition, the changed names are propagated to all ELF files that depend on the entities whose symbolic names were diversified. To experiment with symbol diversification on Raspbian, we employed our old tool that was previously used to diversify x86_64 Linux [18]. To be run on Raspbian, the tool needed to be modified to support 32-bit ELF files, which was relatively straightforward. The necessary source code for the symbol diversification tools in this experiment consists of approximately 2500 lines of C++ code and 300 lines of Python and Bash scripts. We also needed to modify the *glibc* dynamic linker source code with 225 inserted and seven deleted lines in six files, plus an existing third-party SHA-2 implementation that consisted of two files and 1262 lines of code.

The purpose of our experiments with the symbol diversifier was to find our how well it works in a new operating system in the IoT environment and how much manual work is involved in adapting the diversifier into a new operating system. In our experiments, diversified Raspbian was able to successfully start *systemd* and initialize the system. Most of the services in the system started normally, although initially some did not. For example, because of the problems caused by run-time dynamic loading, the login service did not start at first.

To overcome these challenges, we implemented a source code patch to *glibc* that makes it diversify symbols passed to *dlsym* on the fly. The patched *glibc* contains a static data object that contains the diversification secret and the identification number of the algorithm to use. This data can then be rewritten to include the correct diversification secret in the library after compilation and linking, so including the secret is a very inexpensive operation.

With the patched glibc, the symbol-diversified Raspbian can boot, using simple prefixation as the diversification transform. The *login* command works, as does the shell's basic functionality. Programs *less*, *sudo*, *ssh* and *readelf* also worked normally in our experiments. The *ping* command operated normally after we set the *set-user-id* bit to its meta-data. In the case of the *man* program, we had to manually specify the location of the library it needs with the *LD_LIBRARY_PATH* environment variable.

In conclusion, despite some challenges, interface diversification was successful in our experiments. Our memory layout shuffling implementation successfully prevented an exploit from working, and the symbol diversification tool managed to diversify the whole Raspbian operating system. Even though some manual fixes were needed with symbol diversification, IoT operating systems usually have only a limited set of applications, and many nodes in sensory networks are identical, which means these fixes can also made automatically for a number of nodes.

6. Discussion

We have seen that many of devices are currently distributed with identical, well-known interfaces. This software monoculture allows the adversary to devise a single exploit to attack several sensor nodes with identical software. Diversification, by breaking this monoculture, is a promising proactive security mechanism for the widely distributed IoT operating systems.

As the resources of IoT devices are extremely limited, security mechanisms with negligible overhead are needed [68]. This is especially true for sensor networks, where the nodes are often severely limited in terms of the power supply, computational power and memory, for example. In most cases, diversification causes only a very small overhead or no overhead at all. Take system call diversification as an example: Simply replacing the system call mapping with a new one entails no performance penalties, and there are no adverse effects on memory consumption. We also saw earlier that our memory layout diversification scheme has a negligible memory overhead. At the same time, many other traditional protection mechanisms, such as resource intensive anti-virus solutions,

are not such a good fit for low-resource sensor nodes. As a technique that does not consume that many resources, diversification is also highly energy-efficient. Conserving energy is one of the most important challenges in the IoT environment and sensor networks [69]. As diversification has modest resource requirements, it can be argued that diversification provides good security and low energy consumption, which are often seen as conflicting goals.

A huge downside of embedded devices is that they often cannot be updated. When a vulnerability in the software is found, the nodes are exposed to malicious attacks. Although diversification as a security approach does not remove the software vulnerabilities per se, it prevents the adversary from using them. Therefore, diversification alleviates the problem of irregular or absent software updates. Interface diversification is also a proactive security measure in the sense that it provides protection for many zero-day exploits that are currently unknown. Moreover, diversification makes the propagation of malware more difficult, as we have seen. Interface diversified nodes stop malware from moving from node to node in a sensor network.

Still, interface diversification is not a solution to everything. Therefore, it is fortunate that one of the important benefits is what we call orthogonality: diversification can be used in combination with other security measures, such as encryption and authentication mechanisms. It complements other security solutions in the complex IoT environment.

Based on this discussion, it is quite that obvious interface diversification has potential to bring great benefits to IoT systems. However, it also has many limitations as a security measure. First, depending on the obfuscation methods, diversification might introduce some costs for execution time and memory consumption. However, several simple obfuscation methods we discussed in this paper, such as changing the system call numbers and renaming, cause only very modest costs or no costs at all, which makes them perfect for the IoT environment.

Second, the propagation of diversification to keep all parts of the system compatible also poses some challenges. For example, if we diversify the function names in some library, all the applications and other libraries that depend on that library have to be diversified accordingly. We showed in a previous study, however, that this task can be performed automatically with significant accuracy, and help is only rarely required from a programmer [16]. The fact that IoT devices and sensor nodes are usually very limited in terms of the number of applications and the amount of executable code running on them also makes reliable diversification more feasible.

Another challenge associated with interface diversification in sensor networks has to do with the update process. A sensor network may incorporate the capability to share firmware updates between peers instead of only relying on a central source. This way, the relaying nodes should get less overloaded, and therefore, firmware upgrade propagation should be more efficient. From the security point of view, this is no worse than a direct source as long as proper verification mechanisms are in place. However, on a diversified sensor mesh network, updates like these are not possible as each node should essentially have its own unique version of the latest firmware. The architectural possibilities for updating node firmware, therefore, are more limited than on a network free of diversification. A diversified update should be used only on a single node to keep diversification unique, which means there needs to be a repository of $N > M$ unique variations of the firmware, where M is the number of nodes requiring updates in the network. Therefore, one should adjust or at the very least test the network's ability to handle updating during normal operation. To alleviate the bandwidth requirements, nodes can be updated in phases or one at a time. A diversified network should also require a considerably longer time to breach than an ordinary network, because the attacker first needs to find a vulnerability, be able to exploit it and then use brute force to guess the diversification secret before being able to compromise even a single node.

Finally, if diversification is applied to network protocols, the diversification has to be distributed outside one device. That is, all nodes involved have to know and support the diversified protocol, and the diversification secret has to be shared. This is not very practical in big networks, but protocol diversification can still be a useful approach within smaller groups of sensor nodes and in private networks. These challenges and limitations of internal interface diversification and their possible solutions should be discussed more closely in the future work.

7. Conclusions

The growing IoT infrastructure urgently needs novel security approaches to complement its inadequate security measures. When designing security approaches for IoT and for wireless sensor network nodes specifically, it is essential that the limited resources and the available power are taken into consideration while still keeping the security measures effective. In this study, we proposed interface diversification as a solution to this challenge.

Our survey of diversifiable interfaces in IoT operating systems showed that although many of these operating systems still lack commonly diversified interfaces, almost every system has a few potential target interfaces. In addition, there are several promising operating systems, such as Android Things and many Linux-based systems, that are perfect targets for diversification. When the IoT operating systems grow more mature in the future, we can expect to see more diversifiable interfaces in this environment.

We also demonstrated with our proof-of-concept implementations that interface diversification is a feasible security measure that successfully and effectively works on low-resource embedded devices, such as Raspberry Pi, by preventing malicious exploits from running. The whole operating system and software stacks can be diversified automatically, and, although a limited number of manual fixes may be required, the process is usually straightforward.

We believe interface diversification is a promising security measure in the context of the IoT and sensor networks. This is thanks to its low resource requirements, compatibility with other security measures and wide applicability to several interfaces. Unlike some traditional security measures, internal interface diversification is easily transferable to the IoT environment. However, interface diversification requires more research and practical experiments on IoT devices so that it can gain popularity and be rolled out in commodity software.

Acknowledgments: The authors gratefully acknowledge Tekes—the Finnish Funding Agency for Innovation, DIMECC Oy and Cyber Trust research program for their support. This research was also supported by Tekes project 1881/31/2016 “Cybersecurity by Software Diversification”.

Author Contributions: The individual contributions of the authors are as follows. Sampsa Rauti led the writing process and wrote Sections 1,2 and 7 and most of Section 6 and compiled several portions of other sections. Lauri Koivunen wrote Section 3 and portions of Section 6 and worked on extending Section 4. Petteri Mäki had the biggest role in implementing the proof-of-concept tools and wrote the majority of Section 4. Shohreh Hosseinzadeh had a major role in writing Section 5. Samuel Laurén worked on the attack scenarios and participated in implementing the proof-of-concept tools. Johannes Holvitie and Ville Leppänen provided ideas for original research papers and general guidance. Naturally, each author has commented on the paper as a whole.

Conflicts of Interest: The authors declare no conflicts of interest. The funding sponsors had no role in the design of the study; in the collection, analyses or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. International Telecommunication Union. *Overview of the Internet of Things. Recommendation ITU-T Y.2060*; ITU: Geneva, Switzerland, 2012.
2. Gartner. Gartner Says 6.4 Billion Connected Things Will Be in Use in 2016, up 30 Percent from 2015. Available online: http://www.vxdev.com/docs/vx55man/vxworks/guide/c-vm.html (accessed on 23 June 2016).
3. HP Enterprise. *Internet of Things Research Study*; Hewlett Packard Enterprise: Palo Alto, CA, USA, 2015.

4. Kolias, C.; Kambourakis, G.; Stavrou, A.; Voas, J. DDoS in the IoT: Mirai and Other Botnets. *Computer* **2017**, *50*, 80–84.
5. Cohen, F. Operating System Protection through Program Evolution. *Comput. Secur.* **1993**, *12*, 565–584.
6. Forrest, S.; Somayaji, A.; Ackley, D. Building Diverse Computer Systems. In Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI), HOTOS '97, Cape Cod, MA, USA, 5–6 May 1997.
7. Koivunen, L.; Rauti, S.; Leppänen, V. Applying Internal Interface Diversification to IoT Operating Systems. In Proceedings of the 2016 International Conference on Software Security and Assurance (ICSSA), St. Polten, Austria, 24–25 August 2016; pp. 1–5.
8. Mäki, P.; Rauti, S.; Hosseinzadeh, S.; Koivunen, L.; Leppänen, V. Interface Diversification in IoT Operating Systems. In Proceedings of the 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC), Shanghai, China, 6–9 December 2016; pp. 304–309.
9. Steiner, R.V.; Lupu, E. Attestation in Wireless Sensor Networks: A Survey. *ACM Comput. Surv.* **2016**, *49*, 51:1–51:31, doi:10.1145/2988546.
10. Larsen, P.; Homescu, A.; Brunthaler, S.; Franz, M. SoK: Automated Software Diversity. In Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 18–21 May 2014; pp. 276–291.
11. Hosseinzadeh, S.; Rauti, S.; Laurén, S.; Mäkelä, J.M.; Holvitie, J.; Hyrynsalmi, S.; Leppänen, V. A Survey on Aims and Environments of Diversification and Obfuscation in Software Security. In Proceedings of the 17th International Conference on Computer Systems and Technologies CompSysTech'16, Palermo, Italy, 23–24 June 2016; Rachev, B., Smrikarov, A., Eds.; ACM Press: New York, NY, USA, 2016; pp. 113–120.
12. Hosseinzadeh, S.; Rauti, S.; Hyrynsalmi, S.; Leppänen, V. Security in the Internet of Things through obfuscation and diversification. In Proceedings of the 2015 International Conference on Computing, Communication and Security (ICCCS), Pamplemousses, Mauritius, 4–5 December 2015; pp. 1–5.
13. Hosseinzadeh, S.; Hyrynsalmi, S.; Leppänen, V. Obfuscation and Diversification for Securing the Internet of Things (IoT). In *Internet of Things: Principles and Paradigms*; Buyya, R., Dastjerdi, A.V., Eds.; Elsevier: Amsterdam, The Netherlands, 2016; pp. 259–274.
14. Hjelmvik, E.; John, W. *Breaking and Improving Protocol Obfuscation*; Technical Report; Chalmers University of Technology: Göteborg, Sweden, 2010; Volume 123751.
15. Chew, M.; Song, D. *Mitigating Buffer Overflows by Operating System Randomization*; Technical Report CMU-CS-02-197; Carnegie Mellon University: Pittsburgh, PA, USA, 2002.
16. Rauti, S.; Laurén, S.; Hosseinzadeh, S.; Mäkelä, J.; Hyrynsalmi, S.; Leppänen, V. Diversification of System Calls in Linux Binaries. In Proceedings of the 6th International Conference on Trustworthy Systems (InTrust 2014), Beijing, China, 16–17 December 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 15–35.
17. Laurén, S.; Rauti, S.; Leppänen, V. Diversification of System Calls in Linux Kernel. In Proceedings of the 16th International Conference on Computer Systems and Technologies CompSysTech '15, Dublin, Ireland, 25–26 June 2015; ACM: New York, NY, USA, 2015; pp. 284–291.
18. Lauren, S.; Mäki, P.; Rauti, S.; Hosseinzadeh, S.; Hyrynsalmi, S.; Leppänen, V. Symbol Diversification of Linux Binaries. In Proceedings of the 2014 World Congress on Internet Security (WorldCIS), London, UK, 8–10 December 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 74–79.
19. Abrath, B.; Coppens, B.; Volckaert, S.; De Sutter, B. Obfuscating Windows DLLs. In Proceedings of the 1st International Workshop on Software Protection, SPRO '15, Florence, Italy, 19 May 2015; IEEE Press: Piscataway, NJ, USA, 2015; pp. 24–30.
20. Uitto, J.; Rauti, S.; Mäkelä, J.M.; Leppänen, V. Preventing Malicious Attacks by Diversifying Linux Shell Commands. In Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST'15), Tampere, Finland, 9–10 October 2015; pp. 206–220.
21. Uitto, J.; Rauti, S.; Leppänen, V. Practical implications and requirements of diversifying interpreted languages. In Proceedings of the 11th Annual Cyber and Information Security Research Conference), Oak Ridge, TN, USA, 5–7 April 2016; ACM: New York, NY, USA, 2016; Article No. 14.
22. Shacham, H.; Page, M.; Pfaff, B.; Goh, E.J.; Modadugu, N.; Boneh, D. On the Effectiveness of Address-space Randomization. In Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS '04, Washington, DC, USA, 25–29 October 2004; pp. 298–307.
23. Bhatkar, E.; Duvarney, D.C.; Sekar, R. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In Proceedings of the 12th USENIX Security Symposium, Washington, DC, USA, 4–8 August 2003; pp. 105–120.

24. Höller, A.; Rauter, T.; Iber, J.; Kreiner, C. Towards Dynamic Software Diversity for Resilient Redundant Embedded Systems. In Proceedings of the 7th International Workshop on Software Engineering for Resilient Systems, SERENE 2015, Paris, France, 7–8 September 2015; Springer International Publishing: Cham, Switzerland, 2015; pp. 16–30.
25. Tanenbaum, A. *Modern Operating Systems*; Pearson: London, UK, 2014.
26. Liang, Z.; Liang, B.; Li, L. A System Call Randomization Based Method for Countering Code-Injection Attacks. *Int. J. Inf. Technol. Comput. Sci. (IJITCS)* **2009**, *1*, 1, doi:10.5815/ijitcs.2009.01.01.
27. Jiang, X.; Wang, H.J.; Xu, D.; Wang, Y. RandSys: Thwarting Code Injection Attacks with System Service Interface Randomization. In Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems, SRDS 2007, Beijing, China, 10–12 October 2007; pp. 209–218.
28. National Vulnerability Database. Vulnerability Summary for CVE-2014-6271. Initial CVE of Shellshock Vulnerability. Available online: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271 (accessed on 15 December 2017).
29. Portokalidis, G.; Keromytis, A. Global ISR: Toward a Comprehensive Defense Against Unauthorized Code Execution. In *Moving Target Defense, Creating Asymmetric Uncertainty for Cyber Threats, Advances in Information Security 54*; Springer: New York, NY, USA, 2014; pp. 469–480.
30. Xu, H.; Chapin, S. Address-space Layout Randomization Using Code Islands. *J. Comput. Secur.* **2009**, *17*, 331–362.
31. Cook, K. Introduce Struct Layout Randomization Plugin. Available online: https://lwn.net/Articles/723997/ (accessed on 15 December 2017).
32. Shelby, Z.; Hartke, K.; Bormann, C. *The Constrained Application Protocol (CoAP)*; Internet Engineering Task Force (IETF): Fremont, CA, USA, 2014.
33. SYSCALLS.TXT. Available online: https://android.googlesource.com/platform/bionic.git/+/brillo-m9-dev/libc/SYSCALLS.TXT (accessed on 15 December 2017).
34. ChibiOS/RT: Syscalls.c Source File. Available online: http://chibios.sourceforge.net/html/syscalls_8c_source.html (accessed on 15 December 2017).
35. Contiki Wiki. Available online: https://github.com/contiki-os/contiki/wiki (accessed on 15 December 2017).
36. Contiki Homepage. Available online: http://www.contiki-os.org/ (accessed on 15 December 2017).
37. Cloud-Native IoT Operating System for Microcontrollers. Available online: https://github.com/aws/amazon-freertos (accessed on 15 December 2017).
38. INTEGRITY Real-Time Operating System. Available online: http://www.ghs.com/products/rtos/integrity.html (accessed on 15 December 2017).
39. Bhatti, S.; Carlson, J.; Dai, H.; Deng, J.; Rose, J.; Sheth, A.; Shucker, B.; Gruenwald, C.; Torgerson, A.; Han, R. MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.* **2005**, *10*, 563–579.
40. mbed OS uVisor. Available online: https://github.com/ARMmbed/uvisor (accessed on 15 December 2017).
41. Apache Mynewt Repository. Available online: https://github.com/apache/mynewt-core (accessed on 15 December 2017).
42. Baselibc Library—Apache Mynewt. Available online: https://mynewt.apache.org/latest/os/modules/baselibc/ (accessed on 15 December 2017).
43. Nano-RK: A Wireless Sensor Networking Real-Time Operating System. Available online: http://www.nano-rk.org/projects/nanork/wiki (accessed on 15 December 2017).
44. QNX Developer Support. Available online: http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/sys_arch/proc.html (accessed on 15 December 2017).
45. QNX SDP 6.6 Documentation. Available online: http://www.qnx.com/developers/docs/660/index.jsp?topic=%2Fcom.qnx.doc.neutrino.getting_started%2Ftopic%2Fs1_procs_Starting_with_system.html (accessed on 15 December 2017).
46. Memory Configurations. Available online: http://nuttx.org/doku.php?id=wiki:nxinternal:memconfigs (accessed on 15 December 2017).
47. Firmware for Particle Devices. Available online: https://github.com/spark/firmware (accessed on 15 December 2017).

48. Baccelli, E.; Hahm, O.; Gunes, M.; Wahlisch, M.; Schmidt, T.C. RIOT OS: Towards an OS for the Internet of Things. In Proceedings of the 2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Turin, Italy, 14–19 April 2013; pp. 79–80.
49. TinyOS Documentation Wiki. Available online: http://tinyos.stanford.edu/tinyos-wiki (accessed on 22 June 2016).
50. Cao, Q.; Abdelzaher, T.; Stankovic, J.; He, T. The liteos operating system: Towards unix-like abstractions for wireless sensor networks. In Proceedings of the 2008 International Conference On Information Processing in Sensor Networks, IPSN '08, St. Louis, MO, USA, 22–24 April 2008; IEEE: Piscataway, NJ, USA, 2008; pp. 233–244.
51. Wind River Systems. *VxWorks Kernel Programmer's Guide*; Wind River: Alameda, CA, USA, 2003.
52. ZwAllocateVirtualMemory Routine (Windows Drivers). Available online: https://msdn.microsoft.com/en-us/library/windows/hardware/ff566416(v=vs.85).aspx (accessed on 15 December 2017).
53. MMU/MPU Samples—Zephyr Project Documentation. Available online: http://docs.zephyrproject.org/samples/mpu/index.html (accessed on 15 December 2017).
54. Zephyr Project: System Calls. Available online: http://docs.zephyrproject.org/kernel/usermode/syscalls.html (accessed on 15 December 2017).
55. Portokalidis, G.; Keromytis, A. Fast and Practical Instruction-set Randomization for Commodity Systems. In Proceedings of the 26th Annual Computer Security Applications Conference, ACSAC '10, Austin, TX, USA, 6–10 December 2010; ACM: New York, NY, USA, 2010; pp. 41–48.
56. Amadeo, R. Google's new "Android Things" OS hopes to solve awful IoT security. Ars Technica 2016. Available online: https://arstechnica.com/gadgets/2016/12/google-brillo-rebrands-as-android-things-googles-internet-of-things-os/ (accessed on 15 December 2017)
57. Shelby, Z.; Bormann, C. *6LoWPAN: The Wireless Embedded Internet*; John Wiley & Sons: Hoboken, NJ, USA, 2011; Volume 43.
58. Sastry, N.; Wagner, D. Security considerations for IEEE 802.15. 4 networks. In Proceedings of the 3rd ACM Workshop on Wireless Security, Philadelphia, PA, USA, 1 October 2004; ACM: New York, NY, USA, 2004; pp. 32–42.
59. FreeRTOS-MPU—ARM Cortex-M3 and ARM Cortex-M4 Memory Protection Unit Support in FreeRTOS. Available online: https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html (accessed on 15 December 2017).
60. CVE-2016-3714. Available online: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-3714 (accessed on 15 December 2017).
61. Pop, A.R. *DEP/ASLR Implementation Progress in Popular Third-Party Windows Applications*; Secunia Research: Itasca, IL, USA, 2010.
62. Kil, C.; Jun, J.; Bookholt, C.; Xu, J.; Ning, P. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In Proceedings of the 22nd Annual Computer Security Applications Conference, ACSAC '06, Miami Beach, FL, USA, 11–15 December 2006; pp. 339–348.
63. Shacham, H. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS '07, Alexandria, VA, USA, 29 October 2017–2 November 2007; ACM: New York, NY, USA, 2007; pp. 552–561.
64. Crane, S.J.; Volckaert, S.; Schuster, F.; Liebchen, C.; Larsen, P.; Davi, L.; Sadeghi, A.R.; Holz, T.; De Sutter, B.; Franz, M. It's a TRaP: Table Randomization and Protection Against Function-Reuse Attacks. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, It's a TRaP: Table Randomization and Protection Against Function-Reuse, CCS '15, Attacks, CO, USA, 12–16 October 2015; ACM: New York, NY, USA, 2015; pp. 243–255.
65. Diversification on Thingsee One Firmware. Available online: https://gitlab.utu.fi/soft/thingsee-sdk (accessed on 15 December 2017).
66. Shuffle-ld—Layout Shuffling Diversification—Binutils. Available online: https://gitlab.utu.fi/soft/binutils-gdb (accessed on 15 December 2017).
67. Procedure Call Standard for the ARM® Architecture. 2015. Available online: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf (accessed on 15 December 2017).

68. Lopez, J.; Roman, R.; Alcaraz, C. Analysis of Security Threats, Requirements, Technologies and Standards in Wireless Sensor Networks. In *Foundations of Security Analysis and Design V: FOSAD 2007/2008/2009 Tutorial Lectures*; Aldini, A., Barthe, G., Gorrieri, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2009; pp. 289–338.
69. Rault, T.; Bouabdallah, A.; Challal, Y. Energy efficiency in wireless sensor networks: A top-down survey. *Comput. Netw.* **2014**, *67*, 104–122.