SECI - Lightweight Interpreter for Security Algorithms

Stephan Kornemann

IHP

Frankfurt (Oder), Germany

kornemann@ihp-

microelectronics.com

Peter Langendörfer

IHP

Frankfurt (Oder), Germany

langendoerfer@ihp-

microelectronics.com

Oliver Stecklina

Department of Electrical Engineering

and Computer Science, Lübeck

University of Applied Sciences

Lübeck, Germany

oliver.stecklina@fh-luebeck.de

ABSTRACT

Wireless Sensor Networks (WSN), the Internet of Things (IoT) etc. are built using resource constraint devices. The hardware is compiled of different types of micro-controllers and radio frontends on which a plethora of operating systems and protocols is deployed. This poses a huge challenge when developing an intrusion detection system (IDS) that shall be applicable for IoT and WSNs. In order to facilitate such an IDS that is independent of the target platform we propose a security interpreter. In this paper we introduce its concept and architecture and discuss performance parameters such as memory footprint and execution times of different virtualization techniques. Our measurement results indicate clearly that virtualization is feasible. Executing a single instruction takes only 4.2 micro seconds and 1.0 micro seconds in the worst case and the best case, respectively.

CCS CONCEPTS

• **Security and privacy** → **Intrusion detection systems; Systems security; Network security;**

KEYWORDS

IDS, WSN, IoT, security

ACM Reference format:

Stephan Kornemann, Peter Langendörfer, and Oliver Stecklina. 2017. SECI - Lightweight Interpreter for Security Algorithms. In *Proceedings of IoTSec'17, Chennai, India, July 10-14, 2017*, 6 pages. <https://doi.org/10.1145/3084030.3084034>

1 INTRODUCTION

Cyber physical system, wireless sensor networks (WSN) and the Internet of Things (IoT) are going to change our life. Many approaches aim to improve sustainability of production processes, aim health status of human beings and/or buildings etc. The plethora of application fields leads to a broad variety of embedded operating systems such as TinyOS [9], Contiki [3], RIOT [1] and LangOS [19] to name just few of them. The sheer number of network protocols on medium access control (MAC) and routing layer is overwhelming.

The authors of [13] show that attacks against all layers of the network stack exist. This includes jamming attacks on the physical layer and guaranteed time slot (GTS) stealing attacks on the MAC layer [21][17]. On the routing layer there are several attacks such as the sinkhole and wormhole attacks. For most of these attacks, individual methods to detect the tampering have been developed [7][12][18]. For detecting the sinkhole attack several approaches have been proposed [6]. But there is no widely accepted or applicable IDS such as SNORT for the IP world [16]. The reasons why such a solution is still missing are manifold. For our point of view the most two important are:

- the protocols used differ so much in their behavior that using a single approach for describing attack vectors is infeasible
- the operating systems differ so much that there is no way to provide a single implementation of a detection algorithm for all of them.

Current reports about attacks against IoT systems such as traffic lights [20], cars [2] and automation systems [5] show that reliable IDS solutions for this type of systems are definitely needed. In order to reduce the complexity of the development of IDSs we propose to use of virtualization. This allows us to develop a certain algorithm once and to run it then on different platforms. We admit that an adaptation to specific protocol features is still required, but the core logic can be reused. In our vision this approach will allow us to run a distributed IDS relying on a common core logic but being deployed on heterogeneous platforms, where different micro-controllers running different operating systems.

There are several approaches that have dealt with virtualization of sensor applications [10][8][11]. The virtual machines Maté and SwissQM are tightly coupled with TinyOS as host operating system (OS). In contrast we aim a virtualization concept that is independent from the underlying OS. Furthermore, the complexity of the instruction set architecture of these virtual machines is quite limited. Hence, the complexity of applications is limited likewise.

In this paper we focus on the virtualization issue as this is the key enabling technique for our approach and known to be demanded on resource constraint devices. The core contributions of our paper are:

- an architecture for a platform independent security interpreter applicable to a broad variety of WSN operating systems
- a performance evaluation of six virtualization techniques to show that a lightweight virtualization of security tasks is feasible.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

IoTSec'17, July 10-14, 2017, Chennai, India

© 2017 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-5050-1/17/07.

<https://doi.org/10.1145/3084030.3084034>

The rest of this paper is structured as follows. Section 2 introduces the architecture and the concepts of our approach. The Section 3 provides implementation details and measurement results including a performance evaluation of virtualization concepts. The paper concludes with a short summary of our major findings and an outlook on further steps.

2 SECURITY INTERPRETER

The security interpreter (SECI) concept comprises the distribution and execution of security algorithms in the area of IoT. To execute a sole algorithm implementation on different target platforms (hard- and software) virtualization of the execution environment is needed. Thus, SECI implements an interpreter that is based on a virtual instruction set architecture (ISA). At runtime, a translation of the virtual instructions into native operations is performed. Having restrictions and needs in mind an ISA was designed that considers the requirements of security algorithms and deals efficiently with the resources of embedded devices. An applicable architecture requires a tailor-mode ISA that considers the capabilities of the underlayer hardware resources of the low power devices, e.g. the leaking support of floating point operations and some mathematical features. Especially the restricted memory and the energy consumption were considered. SECI uses three types of components to realize an event-based architecture. These components are activities to manage virtual instructions, a file system named SFS to manage events and to load and store data and a scheduler to manage the execution of the activities. The event-based architecture allows a modular design of the security algorithm and thus simplifies its validation.

2.1 Event-driven security algorithms

A typically security algorithm monitors certain parameters and triggers predefined actions in case of reaching or passing a threshold defined for a parameter. The security algorithm reacts on certain event, only. To illustrate our concept we use a simple example, which uses the current packet delivery rate (PDR) and a timer. It checks periodically the PDR and stores the current value in a list. After 60 seconds the average value of all PDR values is computed. Finally, the result is checked against a threshold. If the value below the threshold an alarm signal is generated.

The implementation of this example is split into three parts. The first part collects all PDR values. The second part computes every 60 seconds the average value of the PDR values stored in the list and the third part compares the average value with the threshold value. To integrate these parts into SECI activities that are described in the following subsection are used.

2.2 Activities

An activity is an unstructured container for code and data. The container is used to manage a sequence of virtual instructions. To store internal data a set of registers is provided, which where each activity has its own set of registers. Hence, the context-switching latency can be reduce significantly by storing local state information within the registers. For other data the SECI file system can be used. A description of the file system is given in Section 2.3. Figure 1 depicts three activities. Each of them implements one of

Figure 1: Flow chart of an example security algorithm

Figure 2: State diagram of activities in SECI.

the three parts of our example security algorithm. Each activity includes a prologue which is marked with a dashed box in the figure. The prologue is a sequence of virtual instructions and serves as a precondition that must be fulfilled before the activity can be executed. To predict the execution time of the prologue the code size is restricted. The result of the prologue is a boolean value. If the value is true the state of the activity is set to READY and the activity can be executed. An event is triggered by a SFS write operation. Thus, each activity must be assigned to at least one event. Because of the seamless combination of data and code sequences the security algorithm can react quickly on changes in the system.

Figure 2 shows the state machine of activities. When a new activity is created, memory for its code and its data section are initially reserved and initialized. Afterwards, the activity must subscribe to one or more events. Whenever an event occurs, the prologues of the registered activities are executed. If the result of a prologue is false its activity returns to the state IDLE. Otherwise the activity's state is set to READY and it is added to the scheduler list. The scheduler processes the activities according to the FIFO principle. When an activity is sent to the dispatcher, its virtual code is translated into native code. The execution of an activity cannot be interrupted by another activity. If an error occurs during the execution, the activity is suspended. After completing the execution of an activity its state is switched back to IDLE.

2.3 SECI file system

The SECI file system (SFS) is the core component to communicate between SECI activities and to store and to read data. We defined three different types of data spaces: SYSTEM, PRIVATE and SHARED. The type of the data spaces defines its access rules. The

Table 1: Extract of the SYSTEM segment

Data	Description
PDR	current PDR value from network stack
Timer_1s	Value is updated every second
Timer_1ms	Value is updated every millisecond
NET TX PKT	Number of transmitted network packets

Figure 3: Schematic representation of the accesses to the SFS

PRIVATE section is intended for exclusive use within an activity. Each activity can be assigned to several private segments. For the exchange of data between activities we defined the SHARED segment. The SYSTEM segment is a special memory, which serves as an interface to the OS to get system- and metadata. A selection of the provided data is shown in Table 1. However, the underlying OS must support these system and metadata. If the OS does not support the data a zero as default value is returned.

Activities must be registered to at least one event and an event refers to a memory address in the SYSTEM or SHARED data space. It works like a kind of a watchpoint¹. If a memory address is written, the SFS checks the registered activities and adds the corresponding prologues to the scheduler. Write accesses can be performed by the current activity or by the OS. The OS cannot write directly to the SFS. The OS must call the event handler and the event handler writes the data into the SYSTEM data section of the SFS. Figure 3 shows the control of activities via data accesses.

2.4 Scheduler

The scheduler controls the execution of activities. We use the FIFO principle for our scheduling strategy. This ensures that no race conditions occur. The scheduler has two stages. The first stage is used to manage the execution of prologues and the second stage is used to manage the execution of activities. The exact operating sequence of the scheduler is as follows. The scheduler fetches the first activity from the second stage queue and executes it. If a write access to the SFS is performed, the prologues of the registered

¹Watchpoints are data breakpoints

Figure 4: Flow chart of an example security algorithm

activities for the memory address are added to the first stage queue of the scheduler. It is possible that a currently running activity carries out an additional write access to the same memory address. This would cause that this activity will be listed twice in the first stage list. To prevent it, the list is searched first. After completing the current activity, the first stage list is processed. Then the scheduler returns the control to the OS.

2.5 Example

Now that all parts of SECI are introduced, the example of the security algorithm can be completed. Figure 4 shows the flow chart of an implementation of an example security algorithm. The solid lines shows the control flow of the security algorithm. The OS writes periodically the PDR and the TIMER values to the SFS. The write access triggers the activities, which is indicated by dashed lines in Figure 4. The update of the PDR values triggered Activity 1, which stores the current PDR value in a shared list. The update of the TIMER triggers the prologue of Activity 2. It counts the TIMER calls and if it reaches 60 the average of the PDR list is determined. The result is stored in a SHARED segment. The third activity is executed if the average value is smaller than 40%.

3 IMPLEMENTATION

As the performance of the execution of the virtual instructions is essential for the successful implementation of SECI we discuss six dispatching techniques in this section. Additionally, we explain the integration of SECI into the open source OS langOS [19]. Finally, we present measurement results gathered using a sample application that was executed on a sensor node.

3.1 Dispatching techniques

The dispatcher executes the virtual instructions (VI). To emulate the instructions a dispatching technique is needed. In [22] some

techniques are compared for x86, PPC, MIPS, and ARM architectures. The results show that the performance of every technique depends on the used architecture and on the used compiler features. We are using an MSP430F5438A micro-controller from texas instruments (TI) with a clock rate of 16 MHz [4]. We implemented a short program with VIs that consists of some logic and arithmetic operations. The sequence of instructions is repeated 6,000 times and the total time was measured. Based on these value, we calculate the average time per instruction was computed. For the compilation the GNU MSP430-GCC 5.3 with optimization level `-Os` was used. The measurement results are presented in Table 2.

The fastest technique is based on the return-oriented programing technique (ROP)[15], which is normally used for software attacks. This technique uses the memory addresses of the virtual instructions to emulate a program by forming a sequence of addresses, each VI ends with a return statement. When the start address of the sequence is moved to the stack pointer (SP) and a return statement is performed, the context of the processor is changed and the next VI is executed. The average execution time of $1.0 \mu s$ per instruction enables rapid processing. However, interrupts must be disabled. Otherwise in case of an interrupt the program counter (PC) and the status register (SR) pushed on the stack overwrite the virtual program code.

The second dispatching technique we analyzed, is called direct threading (DT). The basic idea is based on the use of a virtual program counter (VPC). The VPC points to the next VI. When it performs a direct jump to the next instruction the VPC (`jmp @VPC+`) is incremented. Each VI ends with a `jmp @VPC+`, which triggers the next VI. To implement DT the GNU-C compiler extension `labels-as-values` has to be used. The average execution time is $1.37 \mu s$ per instruction. But in contrast to ROP DT can be used with enabled interrupts.

Figure 5: Direct call threading (DCT) vs. indirect call threading (IDCT)

Dispatching methods such as direct call threading (DCT) and DT, need more memory than indirect dispatching methods. Figure 5 shows VIs pointing directly to the memory address of the native code. The MSP430x architecture uses 20-bit, so that storing a single address requires at least 3 bytes and in aligned program code 4 bytes. Having the restricted resource of sensor nodes in mind, this is not acceptable. Furthermore, it is impossible to use direct dispatching techniques in heterogeneous networks because the address of each

Table 2: Comparison of dispatch techniques

Dispatch technique	Time/Instr [μs] @16MHz
ROP dispatching	1.0
Direct threading	1.37
Switch dispatching (optimal)	1.78
Direct call threading	2.2
Indirect threading	2.6
Indirect call threading	3.47
Switch dispatching	4.2

virtual instruction is different on each architecture and for each software version.

To execute the virtual instructions in heterogeneous networks the dispatching technique must be indirect. An indirect technique is the switch-dispatching. It is a very simple and portable solution to realize an interpreter. The needed memory for the addressing is smaller than in direct dispatching techniques. But the performance depends on the location of the VI within the switch statement. In the best case the instructions are located at the beginning of the switch statement, with results in an execution time of $1.78 \mu s$ only. In the worst case the instructions are located at the end of the switch statement, so that an execution time of $4.2 \mu s$ is needed. The difference in the execution time shows clearly that the processing time depends on the structure of the executed code.

As an alternative to the switch-dispatching the indirect call threading (IDCT) can be used. IDCT is a technique which is independent of special compiler extensions and the underlying architecture. Figure 5 shows IDCT using a function table to reference to corresponding virtual instructions which leads to a long execution time. However, IDCT is perfect for SECI because the technique can be used in heterogeneous networks and the processing time is independent of the executed code.

To determine the fastest possible implementation we implemented indirect call threading (IDCT) in assembly code and compared the execution time with the C-code version shown in Listing 1. The used compilers are the C-compiler GNU MSP430-GCC, the TI compiler, and the GNU MSP430 assembler. The results are presented in Table 3.

```

unsigned char program[] = { ... };
void (*itable[]) (void) = { ... };

void indirect_call_threading (void)
{
    unsigned char *vPC = &program[0];

    while (0 != *vPC)
        itable[*vPC++]();
}

```

Listing 1: Source code of an indirect call threading dispatcher

The best result can be achieved by the assembly code. The text segment needs only 24 bytes and the execution time is 24 cycles only. Similar results are achieved by the TI compiler. The worst results are those of the MSP430-GCC implementation. Hence, for implementing SECI we use the IDCT dispatching technique. It is a good trade-off between data size, portability and execution time.

Table 3: Comparison of different compilers for indirect call threading dispatching

Compiler	Cycles	Code size [bytes]
GNU MSP430-GCC 5.3	39	48
TI Compiler 4.4.5	27	32
GNU MSP430 assembler 2.25.2	24	24

3.2 SECI program example

Our example (see Listing 2) shows how the thresholds of a sensor application can be checked by SECI. The application sends periodically a packet with the current temperature and humidity values to a sink. SECI checks the upper and lower thresholds of the sent temperature and humidity values. The assembly code defines a new activity with a register and a code section. The optional *define* section can be used to declare values with a more readable name. To link the activity to a watchpoint the *.event* keyword with the address of the transmitted packet (APP_TX_PKT) is used, so that the activity is executed if a packet is sent to it. The *.register* keyword can be used to define the content of the registers. To check the temperature and humidity thresholds are needed. These thresholds are stored in the registers F0..F3.

The code section contains the application logic. First, register R11 is set to zero and the temperature is read from the payload of the packet. Afterwards, the upper and lower thresholds are checked. This is also done for the humidity value. If the values exceed the pre-defined thresholds a bit is set in R11. Finally, the LED data field is set to the result of the checks.

```
.define
TEMP_VALUE = 2
HUM_VALUE = 6
LED = .0.20 # LEDs
PACKET = .0.16 # APP_TX_PKT
.event PACKET
.register
f0 = 28.0 # max. temperature
f1 = 10.0 # min. temperature
f2 = 60.0 # max. humidity
f3 = 10.0 # min. humidity
r10 = 1 # constant
.code
r11 = 0
f4 = PACKET[TEMP_VALUE]
check0:
if (f4 < f0) goto check1
r11 |= r10
r11 <= r10
check1:
if (f1 < f4) goto check2
r11 |= r10
r11 <= r10
f4 = PACKET[HUM_VALUE]
check2:
if (f4 < f2) goto check3
r11 |= r10
r11 <= r10
check3:
if (f3 < f4) goto end
r11 |= r10
end:
LED = r11
return 0
```

Listing 2: Example of a simple SECI program

3.3 Integration in the langOS operating system

LangOS is a highly configurable system which was designed for low power applications. It is optimized to generate a minimal footprint and to extend life time of low power devices. The configuration process is controlled by configuration files and macros. A preprocessor generates the needed header files and the source files for the build process. A hooking technique allows us to integrated own code in the OS in a very simple way. A hook is a function call that can be placed before or after another function call with the same arguments. To define a hook an entry in the langOS configuration file is needed. This allows us to forward system and metadata from the OS to SECI. The event handler is a component which prepares the meta data and writes them into the SFS. The build chain of langOS automatically includes the event handler of SECI at compile time. For the evaluation, the whole network stack was instrumented. Hooks are placed between the layers app, network, mac and physical (radio). Figure 6 shows the entry points of the hooks. In that way, every incoming and outgoing packet causes an execution of the event handler. The event handler writes the packet data to the SYSTEM section of the SFS. If an activity is registered for this address, the associated prologue is added to the scheduler list. For a real world implementation we used our own sensor node

Figure 6: Integration of SECI into the langOS network stack

platform, which was originally developed for monitoring vital and environmental data of fire-fighters in action [14].

The complete node including the expansion board is depicted in Figure 7. Our sample application uses the SHT21 sensor to measure periodically the current temperature and humidity values. To send the application data to a sink the radio transceiver CC1101 is used. All results presented in this section were measured on this platform.

Figure 7: Used sensor node platform with expansion board

3.4 SECI footprint and timing

To determine the footprint of SECI the sensor application has been implemented for the langOS operating system. It was compiled using the TI compiler version 4.4.5.

Table 4: Size of code and data section of the components in langOS

Component	Code (bytes)	Data (bytes)
langOS & application	28,424	4,736
Dispatcher	436	144
Instructions	760	0
SFS	708	2052
SECI (total)	1,796	2,196

Table 4 shows the size of the data and the code sections of the components. The langOS operating system and the application need 28,424 bytes for the code and 4,736 bytes for the data section. In contrast to the OS SECI has a small footprint of 1,796 bytes for its code and 2,196 bytes for its data section. The major part of the memory (760 bytes) is needed for the virtual instructions. In order to evaluate the performance the execution time of the sample application was measured. For the measurement the sensor values for temperature and humidity values were set to zero. This ensures that the external sensors do not influence the measurements. SECI needs for the execution of the example application only 384 μ s with a clock rate of 4 MHz. The number of executed instructions is 11. This is an average of 34.9 μ s per instruction. With a clock rate of 16 MHz an average of 8.73 μ s per instruction is needed. Compared with the values of Table 2 the execution timer per instruction is 5.26 μ s higher. Fetching the register content was not included in the measurements of the dispatching techniques. We decided to skip the fetch phase to be able to get the pure execution time of an instruction and more significant results.

4 CONCLUSION

Designing security systems especially intrusion detection systems is a tedious and complex task. For systems such as wireless sensor networks and the IoT it is even more tricky than for standard systems as the number of different target platforms and protocols applied is extremely large. In order to cope with this fact we proposed a framework that allows us to run intrusion detection algorithms independent of the underlying platform. Our vision is that this facilitates the implementation of a distributed IDS in which even heterogeneous systems are enabled to cooperate. In this paper we focused on the first step the realization of our security interpreter on a specific system. We discussed its concept and architecture and reported about the integration of SECI into the wireless sensor node operating system langOS. In addition we evaluated different techniques that can be used to realize interpreters on resource constraint devices. Our measurements have shown that executing a single instruction takes only 4.2 micro seconds even for the slowest virtualization technique and 1.0 micro seconds for the fastest one. As virtualization is the core enabling technique for our approach our measurement results indicate clearly that our approach is feasible.

In our next steps we will integrate SECI in at least one more operating system i.e. Contiki. Afterwards, we intend to implement an intrusion detection algorithm that is executed in a heterogeneous environment i.e. on top of langOS and Contiki.

ACKNOWLEDGMENT

The research leading to these results has received funding from the Federal Ministry for Economic Affairs and Energy (BMWi) under grant agreement No. 20K1505B.

REFERENCES

- [1] E. Baccelli, O. Hahm, M. Gunes, M. Wahlisch, and T. C. Schmidt. 2013. RIOT OS: Towards an OS for the Internet of Things. In *Computer Communications Workshops (INFOCOM WKSHPs), 2013 IEEE Conference on*. 79–80. <https://doi.org/10.1109/INFOCOMW.2013.6970748>
- [2] Ch. Miller, Ch. Valasek. 2015. Remote Exploitation of an Unaltered Passenger Vehicle. (Aug. 2015).
- [3] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE, 455–462.
- [4] Texas Instruments. 2015. MSP430F543xA, MSP430F541xA Mixed-Signal Microcontrollers.
- [5] Johannes Klick, Stephan Lau, Daniel Marzin, Jan-Ole Malchow, Volker Roth. 2015. Internet-Facing PLCs - A New Back Orifice, Blackhat 2015,. (2015).
- [6] George W Kibirige and Camilius Sanga. 2015. A Survey on Detection of Sinkhole Attack in Wireless Sensor Network. *arXiv preprint arXiv:1505.01941* (2015).
- [7] Stephan Kornemann, Steffen Ortmann, Peter Langendörfer, and Alexandros Fragkiadakis. 2014. Enabling Wireless Sensor Nodes for Self-Contained Jamming Detection. *Journal of Cyber Security* 3, 2 (2014), 133–158.
- [8] Philip Levis and David Culler. 2002. Mate: A Tiny Virtual Machine for Sensor Networks. (2002).
- [9] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. 2005. Tinyos: An operating system for sensor networks. In *Ambient intelligence*. Springer, 115–148.
- [10] N. Li, Y. Kinebuchi, H. Mitake, H. Shimada, T. H. Lin, and T. Nakajima. 2012. A Light-Weighted Virtualization Layer for Multicore Processor-Based Rich Functional Embedded Systems. In *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. 144–153. <https://doi.org/10.1109/ISORC.2012.27>
- [11] Rene Mueller, Gustavo Alonso, and Donald Kossmann. 2007. SwissQM: Next Generation Data Processing in Sensor Networks.. In *CIDR*, Vol. 7. 1–9.
- [12] E. C. H. Ngai, J. Liu, and M. R. Lyu. 2006. On the Intruder Detection for Sinkhole Attack in Wireless Sensor Networks. In *2006 IEEE International Conference on Communications*, Vol. 8. 3383–3389. <https://doi.org/10.1109/ICC.2006.255595>
- [13] G. Padmavathi and D. Shanmugapriya. 2009. A Survey of Attacks, Security Mechanisms and Challenges in Wireless Sensor Networks. *CoRR* abs/0909.0576 (2009). <http://arxiv.org/abs/0909.0576>
- [14] Krzysztof Piotrowski, Anna Sojka, and Peter Langendoerfer. 2010. Body Area Network for First Responders: A Case Study. In *Proceedings of the Fifth International Conference on Body Area Networks (BodyNets '10)*. ACM, New York, NY, USA, 37–40. <https://doi.org/10.1145/2221924.2221933>
- [15] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.* 15, 1, Article 2 (March 2012), 34 pages. <https://doi.org/10.1145/2133375.2133377>
- [16] Martin Roesch et al. 1999. Snort: Lightweight Intrusion Detection for Networks.. In *LISA*, Vol. 99. 229–238.
- [17] R. Sokullu, O. Dagdeviren, and I. Korkmaz. 2008. On the IEEE 802.15.4 MAC Layer Attacks: GTS Attack. In *2008 Second International Conference on Sensor Technologies and Applications (sensorcomm 2008)*. 673–678. <https://doi.org/10.1109/SENSORCOMM.2008.75>
- [18] Radosveta Sokullu, Ilker Korkmaz, and Orhan Dagdeviren. 2009. GTS attack: An IEEE 802.15. 4 MAC layer attack in wireless sensor networks. *International Journal On Advances in Internet Technologies* 2, 1 (2009), 104–114.
- [19] Oliver Stecklina, Stephan Kornemann, and Andreas Krumholz. 2014. langOS - A Low Power Application-specific Configurable Operating System. In *Proceedings of the 13. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze (FGSN 2014)*. Potsdam, Germany.
- [20] Washington Post. 2015. Can hackers take over traffic lights? (Aug. 2015). https://www.washingtonpost.com/local/could-a-hacker-gain-control-of-dcs-traffic-system/2015/08/08/7cb7cf94-201a-11e5-bf41-c23f5d3face1_story.html
- [21] Wenyan Xu, Ke Ma, Wade Trappe, and Yanyong Zhang. 2006. Jamming sensor networks: attack and defense strategies. *Network, IEEE* 20, 3 (2006), 41–47.
- [22] Mathew Zaleski, Angela Demke Brown, and Kevin Stoodley. 2007. Yeti: a gradually extensible trace interpreter. In *Proceedings of the 3rd international conference on Virtual execution environments*. ACM, 83–93.