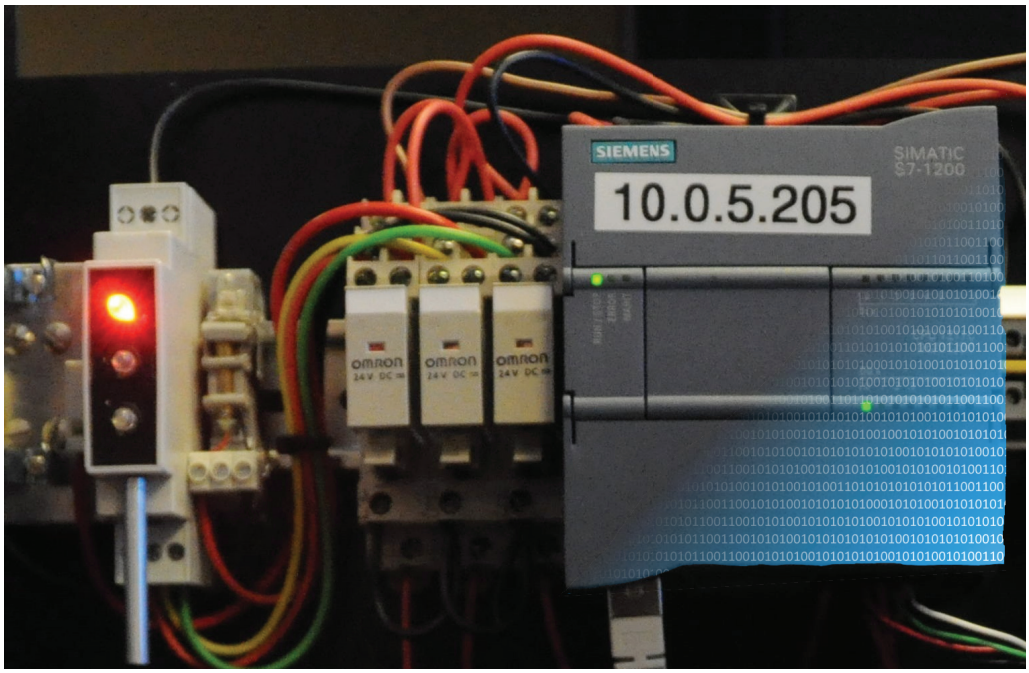
QEMU as a platform for PLC virtualization

An analysis from a cyber security perspective

HANNES HOLM, MATS PERSSON



FOI-R--4576--SE
ISSN 1650-1942
April 2018

Hannes Holm, Mats Persson

QEMU as a platform for PLC virtualization

An analysis from a cyber security perspective

Bild/Cover: Hannes Holm

Titel	QEMU as a platform for PLC virtualization
Title	Virtualisering av PLC:er med QEMU
Rapportnr/Report no	FOI-R--4576--SE
Månad/Month	April
Utgivningsår/Year	2018
Antal sidor/Pages	36
ISSN	1650-1942
Kund/Customer	MSB
Forskningsområde	4. Informationssäkerhet och kommunikation
FoT-område	
Projektnr/Project no	E72086
Godkänd av/Approved by	Christian Jönsson
Ansvarig avdelning	Ledningssytem

Sammanfattning

IT-säkerhetsutvärderingar är ofta svåra att genomföra inom operativa industriella informations- och styrsystem (ICS) då de medför risk för avbrott, vilket kan få mycket stor konsekvens om tjänsten som ett system realiserar är samhällskritisk.

Av denna anledning utforskar många forskare virtualisering som en metod för att möjliggöra realistiska simuleringar av ICS, där IT-säkerhetsutvärderingar kan utföras utan risker.

Ett programmerbart styrsystem (PLC), eller kontroller, är en viktig ICS-komponent som används för att övervaka och styra fysiska processer såsom elkraftbrytare och järnvägsställverk. Tidigare forskning har identifierat PLC:er som en särskilt svår komponent att virtualisera. I denna rapport undersöks möjligheten att virtualisera PLC:er med hjälp av mjukvaran Quick Emulator (QEMU).

Resultaten visar på att det är möjligt att virtualisera PLC:er med hjälp av QEMU. De visar dock också på att implementation av en PLC in QEMU kan vara mycket dyrt, och att IT-angrepp mot den (i QEMU) simulerade PLC:n kan få andra konsekvenser än angrepp mot PLC:n i verkligheten.

Nyckelord: IT-säkerhet, ICS, PLC, QEMU, virtualisering

Summary

Cyber security audits are generally difficult to perform on operational industrial information and control systems (ICS) due to the risk of outages in the often society-critical services realized by these systems.

For this reason, many researchers are exploring virtualization as a means to realize high-fidelity simulations of ICS systems, where cyber security tests can be safely performed.

The Programmable Logic Controller (PLC) is an important ICS component that is used to monitor and control physical processes such as circuit breakers and railroad switches. Previous research has identified PLCs as a particularly difficult component to virtualize. This report explores the possibility to virtualize PLCs using the Quick Emulator (QEMU).

The results indicate that it indeed is possible to virtualize PLCs using QEMU. They however also suggest that an implementation of a PLC in QEMU could be very expensive to produce, and that exploits against the simulated PLC might have different outcome than against the real PLC.

Keywords: Cyber security, ICS, PLC, QEMU, virtualization

Innehållsförteckning

1 Background

The Industrial Information and Control Systems (ICS) that supervise and control most of our critical infrastructures, such as electric power distribution and water distribution, are in the process of transforming from specially constructed technologies to general-purpose Information Technology (IT) solutions. A modern Programmable Logic Controller (PLC) has more in common with a desktop Personal Computer (PC) than its ancestral wired relays, and recent studies have shown that ICS equipment are increasingly being connected to the Internet [1]. This transformation increases functionality and decreases operating costs, but in turn also makes our society more vulnerable to IT attacks. As a consequence, it also puts greater demands on performing IT security evaluations to identify and mitigate threats and vulnerabilities within ICS. Unfortunately, performing IT security evaluations causes stress to systems that can yield unintended problems such as corrupted application states, and in extreme cases even damaged hardware. Such risks are generally not acceptable for ICS as a malfunction could mean power outage for a city or two trains colliding.

This report describes the results from a study within a project called Virtual Industrial Control System (VICS). The purpose of VICS is to evaluate the potential of recreating operational ICS within a virtual environment (often called a “cyber range”), where valid IT security evaluations can be performed without risks. The cyber range CRATE (Cyber Range And Training Environment) managed by FOI in Linköping is used as a baseline for this purpose. The reader is referred to [2], [3], a pre-study of VICS that was carried out during 2015, for a description of virtualization as well as the related concepts of emulation and simulation. This pre-study focused on examining virtualization as a solution for all systems that in one way or another support critical infrastructures. This roughly meant virtualizing equipment within *control centers* (e.g., Supervisory Control And Data Acquisition [SCADA] systems), *communication architectures* (e.g., modems, switches and firewalls), *field devices* (e.g., PLCs), the *physical process* (e.g., a power grid), as well as *business systems*. Of these areas, business systems were left out as most such components already are easily virtualized by common off-the-shelf technologies such as VirtualBox and VMware.

An overview of the implementation methods concerning these areas that were suggested by the pre-study [2], [3] are given in Table 1. The control center and communication architecture were perceived as possible to virtualize without too many issues as they typically are based on publicly available and well documented platforms such as Linux and Windows. The physical process was deemed best simulated, which most ICS vendors already are capable of, and many academics focus on [3]–[7]. Field devices, however, were judged as more difficult to implement in a cyber range. While some field devices are built on standard

technologies that can be virtualized or emulated[1], most are closed and proprietary (e.g., the Siemens S7-1200), and consequently costly to emulate. As an alternative, the pre-study contemplated simulation as a means to enable low-cost implementation of field devices that would satisfy the requirements of some use-cases. The present study, however, focuses on the emulation of field devices.

Table 1. Suggested implementation methods and perceived technical fidelity issues (taken from [3]).

Area	**Implementation**
Control center	Virtualization
Communication architecture	Virtualization
Field devices	Virtualization, emulation, simulation or hardware
Physical process	Simulation

VICS was carried out in collaboration with a project called Automated Vulnerability Assessment (AVA) that was performed by, among others', Idaho National Laboratory (INL) and Draper Laboratories [8]. VICS was funded by the Swedish Civil Contingencies Agency (MSB) and AVA by the Department of Homeland Security (DHS).

Project AVA, in particular Draper Laboratories, spent significant effort on identifying means of emulating field devices such as PLCs. Their documented pre-study [8] identified three methods for accomplishing this objective: (1) using the Quick Emulator (QEMU), (2) using the Low Level Virtual Machine (LLVM), and (3) using QEMU and LLVM in combination. MSB visited DHS in 2016 to discuss the VICS project. During this visit, Draper Laboratories explained that they had continued their efforts based on approach (1) and discontinued (2) and (3). More specifically, they had used a modified variant of QEMU to attempt to emulate field devices. To the authors' knowledge, they managed to partially emulate the Texas Instruments MSP430 microcontroller[2].

The work presented in this paper provides a secondary opinion on the applicability of using QEMU for emulation of PLCs. This is summarized by the two research questions stated below.

- *RQ1*: What steps should be performed to emulate a PLC in QEMU?

[1] For example, the Schneider Electric Modicon Quantum PLC uses an x86 processor, whereas its Ethernet module uses vxWorks 5.4 and a PowerPC processor (MPC870): http://www.digitalbond.com/tools/basecamp/schneider-modicon-quantum/.

[2] https://github.com/draperlaboratory/qemu-msp

- *RQ2*: What effort can be expected to enable emulation of a PLC in QEMU?

Due to resource limitations in combination with the extensive effort on the topic spent by project AVA, analyses of emulation methods other than QEMU were disregarded. Example alternative emulators that were disregarded are:

- **Specialized emulators for certain types of processors**, such as SPIM that emulate the MIPS processor, ARMulator that emulate ARM, and SIMH that emulate old computers like PDP, VAX, IBM and Data General.
- **OVPsim**, a part of the Open Virtual Platform initiative. OVPsim is a multiprocessor platform emulator that can run unmodified production binaries that often are used for emulating parallel computing platforms. It can emulate ARM, PowerPC, MIPS and many other processors, including some FPGA soft processors. The core components of OVPsim are proprietary but free to use. Several of the additional components are open source.
- **Simics** can simulate many CPU architectures such as ARM, x86, MIPS and PowerPC. It was originally developed by the Swedish Institute of Computer Science (SICS), but is now owned by Intel and maintained by Wind River Systems. Simics is most often used to develop software for embedded platforms. An interesting feature of this simulator is that it can run code in reverse direction, which can be quite useful for debugging since you can single step both forward and backwards in the code.

The present study has far fewer resources at its disposal than the study carried out by Draper Laboratories. For this reason, it is out of scope for the study to actually develop a working solution. The purpose is rather to theorize regarding how such an objective could be accomplished as well as estimate the effort involved.

Furthermore, the study only considers PLCs, the arguably most relevant kind of field device. However, its' result will be somewhat applicable to other technically similar devices commonly used as controllers for critical infrastructure applications, such as Remote Terminal Units (RTU).

Finally, the study will delimit from analyzing any legal or policy-related issues that emulating field devices concern.

The remainder of this report is structured as follows: Section 2 provides a technical overview of PLCs. Section 3 describes QEMU. Section 4 describes case studies of two (very) different PLCs and answers the two research questions. Section 5 provides a discussion of the results of the study along some common IT security use-cases involving PLCs. Finally, Section 6 concludes the report and presents future work.

2 Programmable Logic Controllers

During the 1960s, most industrial processes were controlled by a cabinet of hard-wired relay panels. A minor change to the operation of the industrial process required manual rewiring, and debugging faults was time-consuming and costly. [9]

Programmable Logic Controllers (PLC) brought increased automation and flexibility to this labor intensive process. While the first PLCs merely replaced the hard-wired relay logic, they have evolved over the years and today they have more similarities with modern desktop computers than their ancestral wired relay panels.

2.1 Overview of the architecture of a PLC

From an architectural hardware viewpoint, a typical PLC shares many traits with a general desktop computer (or an embedded system in general): it has a processor, a motherboard, a power supply and memory (volatile and non-volatile). Dissimilar to most desktop computers, it has special I/O communication devices that enable monitoring and controlling physical processes. Also dissimilar from most desktop computers, a PLC is built to endure rough environments as well as to function without errors for up to 40 years [10].

An overview of the technical software architecture of a typical PLC is given in Figure 1 [11]. A PLC typically has a real-time operating system (RTOS) that runs applications and drivers; this collection of software is commonly referred to as a *firmware*. A central process running on the RTOS is responsible for communicating with the physical process as well as executing the special relay logic that has been given to the PLC to fit its operating context. This user-created logic is often called the *user program*. The user program is generally developed on a desktop PC, then compiled and transferred to the PLC. Different PLCs support different languages for writing user programs; two common such languages are Ladder logic (LAD) and Statement lists (STL). These are rather simple, low-level languages that have little in common with modern object-oriented software development languages.

In some cases, this core process is also responsible for updating auxiliary services such as fieldbus communication, SCADA, and Human Machine Interfaces (HMI). Such services might include MODBUS, PROFIBUS, BACnet, File Transfer Protocol (FTP), Hypertext Transfer Protocol (HTTP), Secure Shell (SSH), and Telnet. In other cases, auxiliary services might be executed through context switching or even on an entirely separate operating system and hardware. Regardless, the RTOS prioritizes the core relay logic over other applications, thus ensuring that non-critical functions do not delay the execution time of the user program.

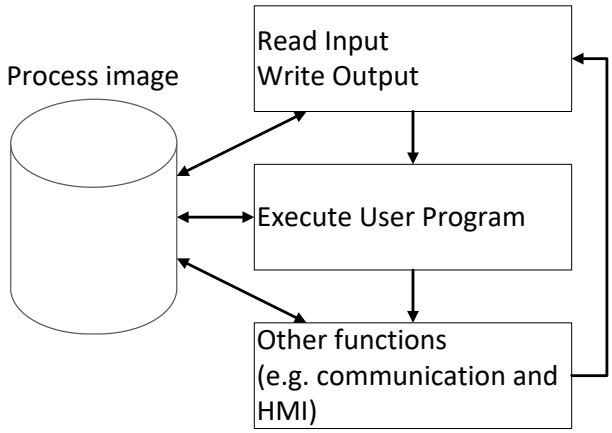


Figure 1. Overview of PLC program execution flow [11].

The core program is executed in so called *scan cycles*, where each cycle has a maximum allowed time limit. If this limit is violated, the program halts and an exception is thrown.

A scan cycle is carried out in the following manner: First, the PLC reads the physical inputs and stores them in the *process image*. The process image stores the states of inputs and outputs; the user program operates on these stored states in the process image. It then executes the user program. The instructions of the user program are structured by Program Organization Units (POU), which commonly include (at least) Organization Blocks (OB), Data Blocks (DB), Functions (FC) and Function Blocks (FB). These instructions are processed in numerical order. Their output values are then written to the process image and to the physical outputs. At the end of each cycle, the process image is refreshed and the resulting outputs are written to the physical outputs.

2.2 Information required to emulate a PLC

In order to emulate a PLC in QEMU (or a similar emulation platform), there is a need to obtain various information regarding the hardware and software of the PLC. This section provides an overview of such information.

2.2.1 Hardware

The processor of the PLC, or more precisely, its instruction set architecture (ISA), needs to be thoroughly understood. This is the case as different processors can have radically different designs that put different requirements on machine code.

For example, the 32-bit version of the x86 architecture (which power many desktop computers) has 12-14 registers, is based on Little Endian and a Complex Instruction Set Computer (CISC) architecture. The 32-bit version of PowerPC, on the other hand, has 32 registers and is based on Big Endian and a Reduced Instruction Set Computing (RISC)[3] architecture. An operating system built for one platform will generally not work for another. If the ISA cannot be identified by vendor documentation or physical inspection, it might be identified by observing machine code compiled for the processor. Such code could be obtained from, for instance, a firmware (see section 4.2.1).

Assuming that an instruction set is completely documented and that this documentation is available in the public domain, it can still be an extremely demanding task to implement it in an emulator. For instance, the instruction set reference for the Intel 64 and IA-32 Architectures consists of 2198 pages [12]. Furthermore, this assumption is generally violated as PLC vendors rarely make their employed processors and their ISA public. In this case, the instruction set needs to be reverse engineered, a generally daunting task.

Auxiliary devices such as Ethernet ports, I/O devices, serial connectors and USB ports generally also need to be properly identified as the drivers of the PLC sometimes are custom-built for specific hardware. That said, fairly standardized common devices such as Ethernet and USB might however already be covered by existing hardware drivers in QEMU.

2.2.2 Software

The first piece of software that needs to be understood is the boot loader, which loads parts of the operating system into memory and bring the system into a by the kernel defined state. The boot loader is sometimes provided by a new firmware used to update the PLC.

Second, there is a need to understand the operating system that is employed, especially concerning the employed kernel. The reason for this is that the operating system and its kernel put compilation requirements on applications that are to be executed on it. For example, an SSH server that has been compiled for Linux Debian will not run as a standard Windows process.

Third, there is a need to understand the device drivers that provide application programming interfaces (API) between programs and the operating system as well as the underlying hardware (e.g., communication devices).

[3] CISC contains more complex machine code instructions than RISC. For example, the one-line CISC instruction “MULT”, which multiplies one value with another, is replaced by four lines of RISC instructions (“LOAD”, “LOAD”, “PROD”, “STORE”).

If the boot loader, operating system and its applications are publicly known and possible to obtain, it is "simply" a matter of correctly identifying (i.e., fingerprinting) them in the PLC. Unfortunately, in the ICS world, these are generally proprietary and unknown. If this is the case, a reasonable solution is to obtain firmwares for the PLC from the vendor[4]. If any available firmware contains a boot loader (more likely on modern PLCs [13]), and this boot loader can be recreated and deployed on an emulated PLC CPU, the firmware could be mounted and installed as if on a real PLC. If no boot loader or firmware can be obtained, there is a need to reverse engineer the software by inspecting a physical PLC [14].

[4] Obtaining PLC firmware is not always simple. For example, to download a Siemens PLC firmware, there is a need to obtain special privileges by personnel at Siemens.

3 QEMU

QEMU, short for “Quick Emulator”, is an open-source virtualization software written in C. The purpose of this chapter is to describe the internals of QEMU as well as how the platform can be modified from a software development perspective. This information was gathered by a combination of QEMU documentation and source code inspection[5]. It is important to mention that there is very scarce developer documentation available. This is well exemplified by a statement in the QEMU developer frequently asked questions section[6]:

> *Question*: Does QEMU have internal documentation?
>
> *Answer*: Not really. The best source of internal documentation are the various KVM Forum talks and potentially talks at other conferences. These can be a bit out dated but often are good overviews of various subsystems.

Furthermore, the existing source code comments seem to generally be directed to developers who already are well accustomed to QEMU development. A typical kind of code comment is exemplified for the method *arm_register_el_change_hook* located in the file *target/arm/cpu.c* (see Figure 2). Apart from the method itself being uncommented, the internal comment within the method does not describe what the method actually does, but rather a limitation within the method with respect to its purpose. Such information is only useful for a developer who already knows the purpose of the method.

```
void arm_register_el_change_hook(ARMCPU *cpu, ARMELChangeHook *hook,
                                 void *opaque)
{
    /* We currently only support registering a single hook function */
    assert(!cpu->el_change_hook);
    cpu->el_change_hook = hook;
    cpu->el_change_hook_opaque = opaque;
}
```

Figure 2. Example QEMU source code comment.

As a consequence, it is a daunting task for a developer to obtain a thorough understanding of QEMU. A thorough understanding is necessary if extensive modifications are desired, such as extending QEMU to support a new architecture.

QEMU has several virtualization modes. First, it can do x86 processor hardware virtualization using Kernel-based Virtual Machine (KVM), executing at almost native speed. Second, it can emulate other processors for use in virtual machines

[5] https://github.com/qemu/qemu
[6] http://wiki.qemu.org/Contribute/FAQ

through real-time translation of machine code. Finally, it can run simple programs for other architectures using real-time translation, similar to Wine in Linux.

QEMU does not have a Graphical User Interface (GUI) and is often used as a part of a more complex virtualization manager. For example, VirtualBox uses parts of QEMU, and Proxmox[7] (another virualization software) uses it for its main virtualization system.

Focusing on PLC emulation, the QEMU system is able to emulate a large range of different processors, for example, x86, ARM, MIPS, Sparc and PowerPC. It can even emulate the less common Tricore processor that can be found in some Siemens PLCs.

QEMU is based on several components (see Figure 3):

- The *Hypervisor* controls the emulation.
- The *Tiny Code Generator* (TCG)[8] translates between guest machine code and host machine code.
- The Software *Memory Management Unit* (MMU) handles memory accesses.
- The *disk subsystem* handles different disk image formats.
- The *device subsystem* handles network cards and other hardware devices.

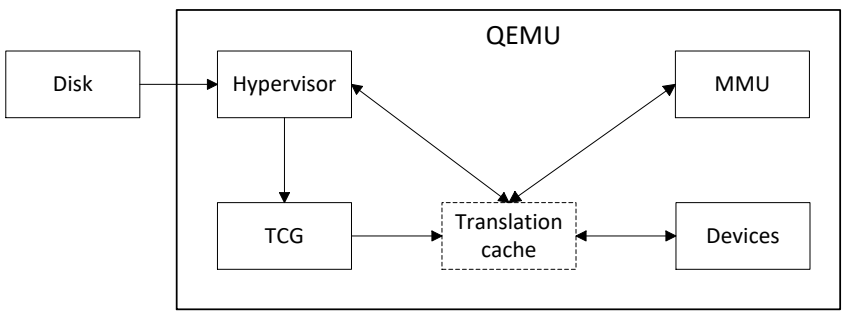


Figure 3. QEMU architecture.

These components are described in the remainder of this chapter.

[7] www.proxmox.com
[8] http://git.qemu.org/?p=qemu.git;a=blob_plain;f=tcg/README;hb=HEAD

3.1 Hypervisor

A hypervisor is a virtual machine monitor that creates and runs virtual machines. The hypervisor in QEMU loads the binary machine code from a disk image, translates it to native machine code with the TCG, connects to virtual or real devices, initiates the software MMU, and then starts emulating the operating system in the disk image.

Alternatively, if the source is x86-code, QEMU can use the KVM feature of the Linux kernel to execute the virtual machine in native mode. KVM is basically a hypervisor inside the Linux Kernel. It can run several operating systems in parallel. QEMU can start a new thread inside KVM in order to execute the emulated OS, and then the KVM takes control of the execution.

3.2 Tiny Code Generator

It is not possible to run machine code compiled for the instruction set architecture (ISA) of one processor on another, for example, ARM machine code on an x86 processor.

In QEMU, the Tiny Code Generator (TCG, see Figure 4) translates source processor machine code to blocks of x86 machine code. The code blocks are put in a translation cache, where they are (if possible) linked together with jump instructions. When the hypervisor later executes the code it can jump into a code block, and the execution can run on different translated code blocks until a new block needs to be translated. In that case it jumps back into the hypervisor which translates more code with the TCG.

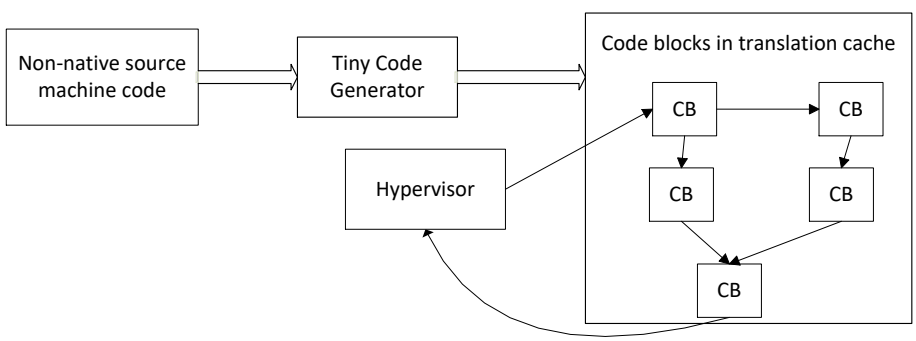


Figure 4. Overview of Tiny Code Generator.

The TCG has a minor disadvantage from a general use-case, which becomes a major disadvantage from an IT security viewpoint: It is not able to correctly run self-modifying code since it doesn't mark the modified code page as "dirty" and in need of re-translation[9]. Self-modifying code is quite rare in the software world, but is used by many kinds of exploits. In particular, memory corruption exploits such as buffer overrun, which overwrite part of the vulnerable application with special code provided by the threat agent (e.g., a back door), fail if the code that is overwritten already has been run (and thus cached) by the TCG. This makes debugging such exploits very difficult and provide false negatives during security audits. Memory corruption exploits constitute one of the most common as well as problematic kinds of attacks as they typically are able to provide the privileges of the vulnerable application.

It is possible to program the TCG to handle new types of processors. This is not too demanding if there already exists a similar processor from which code (translation methods) can be copied. However, if the new processor uses more registers than an x86 processor and has many complex instructions, extensive effort is probably required. Luckily, many processors often have similar instructions. For example, the "MOV" instruction exists in almost all processors, and can simply be copied, unless there is some bit size difference in CPU registers. For instance, emulating a 64-bit processor on a 32-bit processor can require many extra instructions, which also take more time to program in the TCG translator.

Inside the source code of QEMU there is a subdirectory called '*tcg*', which contains code that translates machine instructions to the corresponding x86 machine instructions. This code is a simple translation state machine written in C. There are also special translations for memory access and jumping, because these can generate calls to the software memory management unit. This is important as QEMU protects other memory areas outside the code blocks. The jumps and branching in the machine code must also arrive at the correct memory address.

3.3 Hardware devices

Hardware device requirements by a virtual machine can be fulfilled either by directly connecting real physical devices in the host, or by hardware device emulation in QEMU. Most of the QEMU code related to auxiliary hardware is located in the directory "*hw*".

The direct connection to devices is sometimes called "pass-through". This means that the virtual machine gets direct access to the USB bus, or the PCI bus, and can communicate directly with the device. This can be required by, for example, webcams and serial- and parallel ports. Some devices are difficult to share with

[9] https://github.com/unicorn-engine/unicorn/issues/820

the host, for example network devices. This is solved by emulating a network card, hereby adding an extra layer on the networking stack, which slows down the network speed. QEMU can alternatively connect to the “virtio” paravirtualization drivers in the Linux kernel, which means that the Linux kernel handles input/output between the virtual machine and the hardware devices instead of QEMU (which merely acts as a mediator).

3.4 Disk images

QEMU can handle several different disk image formats. The preferred formats are *raw* or *qcow2*. Raw is a very simple format that stores the bytes in the filesystem on a file, just byte for byte. This format is supported on most other emulators. Qcow2 is QEMUs own image format and is useful for small images. It also has support for disk image compression as well as capturing snapshots of a disk image state. Two other formats are also supported: *vdi* which is used in VirtualBox and *vmdk* which is used in VMWare.

3.5 Software MMU

The Memory Management Unit (MMU) in traditional processors handles the access to computer memory locations. When the processor wants to access a certain memory address, the MMU fetches the content of that address. This content can come from a local fast cache on the processor chip, from Random Access Memory (RAM), or from disc. It can even make some smart decisions about where to cache certain memory locations.

QEMU has a software based MMU that works similarly to a hardware MMU. It uses an address translation cache which contains the guest addresses, host addresses and offset values to increase translation speed. It can also allow for a smart chaining of code blocks, in order to enable faster execution without memory faults where it must reload and retranslate memory blocks.

When looking for exploits for a PLC running in QEMU, there might be some uncertainties about whether the software MMU is translating and placing the blocks correctly.

4 Results

This chapter first presents two case studies of PLCs focusing on emulation in QEMU (Section 4.1). It then presents a conceptual method for emulating a PLC in QEMU under different circumstances. The steps involved in this method are described in Section 4.2. The effort required to accomplish these steps are discussed in Section 4.3.

4.1 Case studies

Most PLC brands (and even models) are very dissimilar from a technical viewpoint [3]. The two case studies presented in this section should be read with this in mind.

4.1.1 Siemens S7-400/1200

FOI has conducted technical studies of the Siemens S7-400 and S7-1200 models on previous occasions [3], [15]. The technical aspects of relevance to QEMU emulation from these studies are described in this section.

Siemens S7-400 employs two Infineon TriCore processors. TriCore processors are officially supported by QEMU, however, it is unknown whether this architecture has been specially modified by Siemens. If this is the case, there is a need to build a new target in QEMU that extends the existing QEMU TriCore configuration. The S7-400 runs a kind of virtual machine on top of the TriCore architecture. For example, a long unconditional jump in a TriCore ISA begins with the machine code byte 0x1d, and a similar unconditional jump inside the PLC virtual machine begins with the machine code bytes 0x70 0x0b. The machine code inside the virtual machine is called MC7 (Machine Code 7). MC7 is undocumented and has not yet been completely reversed by external parties [15]. However, it is hopefully not necessary to completely reverse MC7 to enable emulation. A simpler solution would be to implement the hardware ISA (i.e., extend the QEMU TriCore target if necessary) and completely clone a S7-400 firmware to a QEMU image. Due to the lack of documentation, it would be beneficiary to obtain a clone that includes a complete firmware, including boot loader and a complete RTOS. Apart from the CPU and ISA, it is unknown whether profiles for auxiliary hardware devices of the S7-400 such as Ethernet and I/O need to be implemented in QEMU.

The S7-1200 employs a completely different machine code than the S7-400, and the exact hardware CPUs within the S7-1200 have not been identified [15]. The S7-1200 employs a virtual machine similar to that of S7-400, but with different machine code than MC7. Reversing the virtual machine code is also more difficult for S7-1200 as there is no documented high-level assembler representation available.

To sum up, Siemens S7-400, which has been widely researched and reversed by the community, would be very expensive to attempt to emulate in QEMU. S7-1200 would be even more expensive, as its properties are largely unknown to external parties.

4.1.2 OpenPLC

OpenPLC is an open-source implementation that emulates a PLC on Linux[10] that is available on GitHub[11]. Implementing OpenPLC in QEMU is very straightforward as it easily can be compiled and installed on Linux and an x86 architecture. In fact, less than a work-day was required to obtain a functional QEMU emulation.

4.2 Conceptual method

This section answers RQ1, "*What steps should be performed to emulate a PLC in QEMU?*". Emulation using QEMU on overall is illustrated by Figure 5 and requires implementing hardware properties as well as software properties. Hardware requirements are discussed in Section 4.2.1; software requirements are discussed in Section 4.2.2.

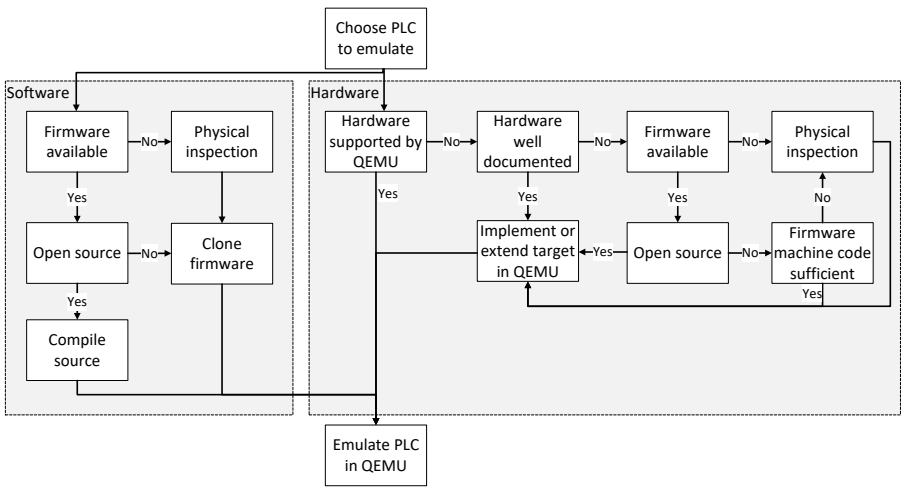


Figure 5. Conceptual method for emulation of PLCs in QEMU.

[10] http://www.openplcproject.com/
[11] https://github.com/thiagoralves/OpenPLC and https://github.com/thiagoralves/OpenPLC_v2

4.2.1 Implementation of hardware

If all PLC hardware devices already are supported by QEMU, the base requirements for emulation are already satisfied. However, it is possible that at least one or more necessary components (e.g., the ISA or an I/O device) either are completely missing from QEMU or requires adjustment. If this is the case, manual effort is required to implement the component or extend an existing component. Well-described technical documentation of the hardware can provide sufficient information for the development of an emulated equivalence in QEMU. As described in Section 2.2.1, it is however probable that adequate technical specifications cannot be obtained.

Given this scenario, an obtained PLC firmware could be used to extract information about the hardware that it is supposed to be deployed on. If the firmware is open source (a very uncommon scenario for PLCs), it can be assumed that the source code contains sufficient information about the supported hardware to enable development of an emulation target. In other cases, packed or compiled PLC firmware are generally available on the Internet by their vendors (although special access restrictions often apply). Compiled machine code in a firmware can be used to both identify (and in some cases even reverse) the ISA, and drivers within the firmware can be used to identify which auxiliary hardware that are employed. In the best case, hardware addresses used by I/O are revealed by the firmware, and general hardware, such as Ethernet (RJ45) ports, are already covered by QEMU.

If the firmware cannot be obtained, there is a need to physically inspect the desired hardware. The nature of this physical inspection depends on the properties of the PLC. For example, in some cases its firmware might be extracted through FTP or a portable media; in other cases, it might be necessary to extract and study its non-volatile memory (e.g., ROM or flash memory).

4.2.2 Implementation of software

To produce a valid QEMU image, there is a need to obtain and install a boot loader and a firmware that correspond to the desired PLC configuration. In the odd case where the desired bootloader and firmware are available and open source, it is generally a simple task to compile them and build a valid image. If a packed/compiled firmware update can be obtained from a vendor, it could be used to prepare a corresponding image. Such a firmware update could be complete in terms of functionality, or only patch some designated code. A firmware update can also come with or without a boot loader. Furthermore, a firmware update can be as simple as copying all its code in sequence to the file system, and as complicated as unpacking and altering its code to different locations in the file system during extraction. In the simplest of cases, an obtained closed firmware can be directly cloned to an appropriate file system and mounted in QEMU. In a more complicated

case, various reverse engineering techniques are required to recreate an extracted firmware and bootloader. In a very complicated case, physical inspection is necessary to understand and clone both the boot loader and the firmware.

4.3 Implementation effort

This section answers RQ2, "*What effort can be expected to enable emulation of a PLC in QEMU?*". In the general case, it is next to impossible to accurately predict the effort required to emulate a PLC using QEMU. This is because the general effort required to reverse engineer machine code is unknown, and the effort required to develop code is uncertain. Each activity in Figure 5 will simply have radically different results depending on the exact use-case. The reader should view the estimates provided in this section with this in mind.

Given the simplest of scenarios, where all PLC hardware are directly supported by QEMU, and a firmware can be obtained as source code or simply cloned to a system image (i.e., little or no reverse engineering of machine code is required), perhaps as little as a work-day would be sufficient to create an emulation.

Two more complex topics concern the effort required to i) reverse engineer machine code to understand hardware and software, and ii) implement a new architecture in QEMU when the hardware is understood. These two topics are discussed in Section 4.3.1 (i) and Section 4.3.2 (ii).

4.3.1 Reverse engineer machine code

There are next to no data available on the effort required to *understand* software code in general, and machine code in particular. The present research could in fact only find one study that has empirically studied this for machine code. That is, Johnson and Ekstedt [16] studied the effort required to comprehend C and assembly language for the x86 architecture. The authors' presented program instructions corresponding to a simple bubble sorting program written in C and assembly to five engineering students. Each instruction prompted a response from the student that was thought to represent an understanding of the instruction. For example, the instruction "MOVL W, 3" required the student to first type W and then 3 on the keyboard. The results showed that the assembly version of the program took 11 minutes and 29 seconds to understand, whereas the C program took 3 minutes and 44 seconds to understand. If disregarding the low sample size and the question of whether or not the study measured actual code understanding (rather than simply matching instruction types to keypresses), roughly 12 minutes to click through 53 lines of assembler code does provide some insight into the difficulty of reverse engineering machine code.

There are some data on the related topic of software vulnerability discovery in compiled machine code and exploit development for such vulnerabilities. Both of

these topics require the researcher to perform at least some reverse engineering of machine code. Sommestad et al. [17] studied the effort required to discover a zero-day vulnerability in compiled machine code through estimates by experts. The experts estimated between 0 and 855 work days to discover a single zero-day. In a similar study, Holm et al. [18] studied estimates by experts concerning vulnerability discovery in software built using interpreted languages such as PHP and Python (i.e., not machine code). The consulted experts estimated a time between 1 and 53 hours to discover a zero-day. These results illustrate the difficulty and high variability of the reverse engineering process.

4.3.2 Effort to add a new architecture to QEMU

While there is limited data on the actual effort by an individual that is required to develop software code in different languages, it has been significantly more researched than the topic of reverse engineering. A literature review of software development cost estimation studies by Jørgensen and Shepper [19] identified 304 papers in 76 academic journals. Of these, 65 papers provided data obtained from either surveys (27 papers), experiments (19 papers), case studies (8 papers) or real-life evaluations (11 papers).

Research on the effort required to develop software has resulted in various models of different complexity. Of these, regression-based methods dominate, in particular, the Constructive Cost Model (COCOMO) model [19]. COCOMO consists of a set of equations that enable estimating the effort (in man-months) that are required to develop software.

The present study employed COCOMO to estimate the effort required to develop a new architecture in QEMU by inspecting the source code of the QEMU main branch in Github as well as all Github forks of QEMU. Code related to different hardware architectures were identified through manual inspection. In the current version of QEMU, this code could mostly be mapped to the directories "*hw*", "*target*", "*linux-user*" and "*tcg*". Older versions of QEMU do not contain the directory "*target*". Instead, each target has its own directly/path in the project root beginning with the prefix "*target-*", For example, ARM has the path "*target-arm*" in older QEMU versions and "*target/arm*" in the current QEMU version.

As of the 28th of February, 2017, there were 1188 forks[12] of QEMU in Github[13]. A script was used to identify all forks that had developed support for previously unsupported targets, as well as estimate the costs required to develop these targets[14]. Of the 1188 forks, only four hade developed new architectures in QEMU. Of these four architectures, one implementation seems to only have been started

[12] Development of a revised/new software based on some existing software source code.
[13] https://github.com/qemu/qemu
[14] Using the API of Github, https://api.github.com

(ARC4 by zuban32/qemu, which contained a mere 81 lines of code). The remaining three were somewhat completed/working according to their documentation.

An overview of the 20 currently supported architectures in QEMU as well as the four architectures added by forks of QEMU is given by Table 2. The lines of code (LOC) required to implement an architecture greatly differs, from ~1200 for Moxie to ~67600 for ARM. As PLCs use architectures of various complexity, it can thus be expected that new architectures require between a few months to several years to develop. Similarly, the effort required to alter an existing architecture depends on the complexity of the architecture as well as the magnitude of the revision.

Table 2. Effort required to develop a QEMU target.

Architecture	LOC	Effort (months)	Source
arm	67644	200.43	qemu/qemu
ppc	63280	186.87	qemu/qemu
i386	46261	134.49	qemu/qemu
mips	40782	117.82	qemu/qemu
s390x	26909	76.14	qemu/qemu
meta	23089	64.83	img-meta/qemu
sparc	15406	42.39	qemu/qemu
tricore	12444	33.88	qemu/qemu
m68k	7966	21.21	qemu/qemu
bfin	7848	20.88	vapier/qemu
xtensa	7749	20.60	qemu/qemu
cris	6898	18.23	qemu/qemu
sh4	6433	16.95	qemu/qemu
alpha	6415	16.90	qemu/qemu
hppa	4955	12.88	qemu/qemu
tilegx	4783	12.41	qemu/qemu
microblaze	4159	10.72	qemu/qemu
openrisc	3662	9.38	qemu/qemu
unicore32	3596	9.20	qemu/qemu
lm32	2832	7.16	qemu/qemu
nios2	2518	6.33	qemu/qemu
msp430	1372	3.35	draperlaboratory/qemu-msp
moxie	1244	3.02	qemu/qemu
arc4	81	0.17	zuban32/qemu

5 Emulating a PLC: Overkill or too limited?

This chapter explores three use-cases that PLCs might have from a cyber security perspective, and how an emulation in QEMU could fulfill the requirements of these use-cases. These use-cases are: vulnerability discovery (Section 5.1), education and training (Section 5.2), and honeypots (Section 5.3). A summary discussion of these use-cases, as well as an alternative to emulation in QEMU, is presented in Section 5.4.

5.1 Use case 1: Vulnerability discovery

The perhaps most obvious use-case of PLCs from a cyber security perspective is to discover and mitigate novel vulnerabilities within them (see e.g. [15]). As most PLCs are proprietary devices, this activity traditionally involves physical inspection of real PLCs. Obtaining a physical PLC from a vendor for the purpose of vulnerability discovery is, however, not a simple endeavor. One problem is that vulnerability discovery generally involves administrating malformed commands to the target that hopefully triggers an error; an indicator of a potential vulnerability. It is possible that such an error in extension permanently damages the PLC firmware, or even causes a physical malfunction. Another problem is the potential economic damage to the PLC vendor as a result of vulnerability disclosure [20].

Thus, from a practical viewpoint virtualization would certainly make it easier for security researchers to identify vulnerabilities. If one disregards the fact that emulation in QEMU is expensive, there are still questions regarding the validity of such an emulation that need be considered. As discussed in Section 3.2, the conversion between VM and host machine code through the TCG could cause some exploits that in reality would work to fail against the emulation. Other aspects regarding the emulation (e.g., the MMU) could have similar unforeseen impact on the vulnerability discovery process.

As exploit debugging related to inaccuracies in the emulation would be very expensive, it could actually be simpler (and less expensive) to obtain a real physical PLC.

As an alternative, one could attempt to identify any aspects of QEMU that could impact the validity of an emulation from a cyber security perspective, such as the TCG code cache. However, QEMU’s complex code-base in combination with the lack of developer documentation suggests that this would not be a simple endeavor.

5.2 Use case 2: Education and training

A second common use-case of PLCs concern education and training: to teach students how to attack and defend PLCs. Different kinds of education objectives place different requirements on the validity of a PLC emulation. For example, lab sessions that teach IT security novices how PLCs communicate or respond to port scans only require that the emulations behave *similar* to real PLCs. In this scenario the student would not be able to distinguish between a common Linux distribution that has been especially prepared to behave as a Siemens S7-300 PLC and a real physical S7-300. In other words, a QEMU emulation would be a significant waste of resources. An exercise that involves security forensics of a PLC might on the other hand require physical inspection (i.e., the actual physical device), with a QEMU emulation being insufficient. Employing a QEMU emulation of a PLC for an exercise that involves exploitation of PLC vulnerabilities is from one viewpoint overkill and another viewpoint insufficient: It could be overkill as the effect of exploitation could (for novice students) be simulated by a honeypot; it could be insufficient as the exploit might not behave as it would against a real PLC.

5.3 Use case 3: Honeypot

A third use-case of a PLC is as a so called Honeypot; a kind of trap that serves to deceive and detect threat agents. This trap could either be located in a real operative network to warn security analysts of successful intrusions, or in a lab to collect data regarding the behavior of attackers.

To deceive an expert attacker, a QEMU emulation is likely a “lowest bar”. It is possible that it is insufficient, as the timings of different operations made by the emulated PLC likely often would differ from those of its real counterpart. Also, any artifacts in the VM that reveals that it is emulated should naturally be removed. This is not always simple with modern virtualization solutions as tools inside a VM are necessary to enable many vital external monitoring and control applications, for example, the VirtualBox guest additions.

To deceive a novice attacker, a QEMU emulation is likely not worth the implementation effort as the novice would not be able to distinguish from a somewhat decent simulation and an emulation in QEMU.

5.4 An alternative to QEMU

As previously described, a QEMU emulation of a PLC might be insufficient for vulnerability discovery and education/honeypots involving experts, and overqualified for education/honeypots involving novices.

An alternative when QEMU is insufficient is to employ a real physical PLC. An alternative to when QEMU is overqualified is to employ a simulator. A prototype of such a simulator was developed and tested by a master thesis carried out as a collaboration between VICS (FOI) and the Royal Institute of Technology (KTH) [3]. This prototype was based on a standard Linux kernel and the software Honeyd, which is a program that simulates a complete IP stack and thus can deceive network scanners by replying to network packets as a PLC. The prototype extended Honeyd by also implementing ModBus TCP through an open-source Java library. This enabled deceiving novices who would use network sniffing tools for fingerprinting.

A more advanced simulator could include more common PLC software, such as HMI functions through telnet and HTTP. It could also include signature detection for known exploits against the simulated PLC. If such an exploit is used, the PLC could simulate a successful compromise by providing the threat agent with a simulated back door experience (e.g., using an especially written service).

6 Conclusions and future work

This study explored the expected method (RQ1) and effort (RQ2) required to emulate PLCs in QEMU for cyber security purposes.

The overall conclusions regarding RQ1 is that it certainly is possible to emulate most (if not all) PLCs in QEMU. The actual method would however greatly depend on the PLC in question.

The overall conclusions regarding RQ2 is that the effort required to implement a PLC in QEMU, similar to method, greatly varies depending on the context. The general PLC is, however, expected to be expensive to implement in QEMU. First, most PLCs are proprietary and thus expensive to reverse. Second, if a PLC is completely reversed, it is likely that QEMU needs to be extended to enable emulation. As QEMU is a low-level software that has next to no developer documentation and sparse (from the perspective of a new QEMU developer) relevant code documentation, the implementation alone is expected to be taxing. This is exemplified by the mere three added targets by the ~1200 forks available in Github.

Furthermore, to fulfill accurate emulations from a cyber security perspective, there is a need to further study (and possibly alter) the TCG and other internal QEMU aspects as these could alter the behavior of exploits. Exploits that would work in reality might fail against the QEMU emulation; exploits that would fail in reality might enable escaping from the QEMU emulation into the host.

A less expensive alternative to QEMU that could fulfill the expectations of education and honeypots that target novices is to employ a simulator. An alternative to QEMU for vulnerability discovery and education/honeypots targeting experts is to employ physical PLCs.

7 References

[1] H. Holm, “NCS3 - Internetanslutna styrsystem i Sverige (FOI-R--4415--SE),” Linköping, Sweden, 2017.

[2] H. Holm, M. Karresand, A. Vidström, and E. Westring, “A Survey of Industrial Control System Testbeds.,” in *NordSec*, 2015, pp. 11–26.

[3] H. Holm, M. Karresand, A. Vidström, and E. Westring, “Virtual Industrial Control System Testbed (FOI-R--4073--SE),” 2015.

[4] S. R. Nassif and J. N. Kozhaya, “Fast power grid simulation,” in *Proceedings of the 37th Annual Design Automation Conference*, 2000, pp. 156–161.

[5] K. Mets, J. A. Ojea, and C. Develder, “Combining power and communication network simulation for cost-effective smart grid analysis,” *IEEE Commun. Surv. Tutorials*, vol. 16, no. 3, pp. 1771–1796, 2014.

[6] M. M. Fioroni, J. G. Quevedo, I. R. Santana, L. A. G. Franzese, D. Cuervo, P. Sanchez, and F. Narducci, “Signal-oriented railroad simulation,” in *Proceedings of the 2013 Winter Simulation Conference: Simulation: Making Decisions in a Complex World*, 2013, pp. 3533–3543.

[7] J. M. Wagner, U. Shamir, and D. H. Marks, “Water distribution reliability: simulation methods,” *J. water Resour. Plan. Manag.*, vol. 114, no. 3, pp. 276–294, 1988.

[8] I. N. L. (INL), “Control system automated vulnerability assessment study,” 2013.

[9] K. T. Erickson, “Programmable logic controllers,” *IEEE potentials*, vol. 15, no. 1, pp. 14–17, 1996.

[10] Y. Sun, T. Ma, B. Huang, W. Xu, B. Yu, and Y. Zhu, “Risk assessment of power system secondary devices for power grid operation,” in *Electricity Distribution (CICED), 2012 China International Conference on*, 2012, pp. 1–5.

[11] R. Spenneberg, M. Brüggemann, and H. Schwartke, “Plc-blaster: A worm living solely in the plc,” *Black Hat Asia, Mar. Bay Sands, Singapore*, 2016.

[12] Intel, “Intel® 64 and IA-32 Architectures Software Developer’s Manual Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z,” 2016.

[13] J. Zaddach and A. Costin, “Embedded devices security and firmware reverse engineering,” *Black-Hat USA*, 2013.

[14] I. N. L. (INL), “Control System Automated Vulnerability Assessment Study,” 2013.

[15] A. Widström, “FOI-R--4029--SE, Möjligheter och problem vid analys av

fientlig kod riktad mot Siemens S7-serie," 2012.

[16] P. Johnson and M. Ekstedt, "Predicting the effort of program language comprehension: The case of HLL vs. Assembly." KTH Royal Institute of Technology, 2005.

[17] T. Sommestad, H. Holm, and M. Ekstedt, "Effort estimates for vulnerability discovery projects," in *Proc. of 45th Hawaii International Conference on System Sciences*, 2012, pp. 5564–5573.

[18] H. Holm, M. Ekstedt, and T. Sommestad, "Effort estimates on web application vulnerability discovery," in *46th Hawaii International Conference on System Sciences*, 2013, pp. 5029–5038.

[19] M. Jorgensen and M. Shepperd, "A systematic review of software development cost estimation studies," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, 2007.

[20] R. Telang and S. Wattal, "An empirical analysis of the impact of software vulnerability announcements on firm stock price," *IEEE Trans. Softw. Eng.*, vol. 33, no. 8, pp. 544–557, 2007.

FOI, Swedish Defence Research Agency, is a mainly assignment-funded agency under the Ministry of Defence. The core activities are research, method and technology development, as well as studies conducted in the interests of Swedish defence and the safety and security of society. The organisation employs approximately 1000 personnel of whom about 800 are scientists. This makes FOI Sweden's largest research institute. FOI gives its customers access to leading-edge expertise in a large number of fields such as security policy studies, defence and security related analyses, the assessment of various types of threat, systems for control and management of crises, protection against and management of hazardous substances, IT security and the potential offered by new sensors.

FOI
Swedish Defence Research Agency
SE-164 90 Stockholm

Phone: +46 8 555 030 00
Fax: +46 8 555 031 00

www.foi.se