Computing has made its way into most of our lives as a key processor of vast quantities of information. This has happened directly in terms of gadgets and devices that assist us in everyday life, but also indirectly, through the critical infrastructures that enables these devices to function. A key issue with critical infrastructures such as transportation, communication, power-grids and finance, is increasingly circular interdependencies. Because of this issue, a disruption in either one can cascade and have a global effect on the others. To manage these complexities, we are depending on a number of monitoring systems that allow operators and other stakeholders to, within their respective expert domains, discover disruptions as early as possible and then take appropriate actions.

These monitoring systems are not without challenges of their own. In addition to having evolved organically alongside their respective infrastructures, there is a considerable legacy to account for, with both hardware and software components spanning decades of computing history. This puts heavy restrictions on the kinds of interventions that can be performed safely, implying that these systems are ill fit for handling the software and software security landscapes of today, where updates and adjustments need to be applied on a daily basis in order to stand a fighting chance.

The work presented herein address some of the major challenges in securing these monitoring systems against current and future threats posed by antagonistic actors, dormant software defects and changes imposed by technological advances and academic discoveries. This is approached on several fronts in parallel: by embedding resilience in order to allow for controlled experimentation and evaluation of new protection mechanisms in incrementally sensitive settings; by developing laboratory facilities for resilient smart power-grids; and by developing tools and training scenarios for operators of adaptive and reconfigurable monitoring systems.
Monitoring Infrastructure Affordances

Björn Ståhl
Monitoring Infrastructure Affordances

Björn Ståhl

Doctoral Dissertation in Computer Science

School of Computing
Blekinge Institute of Technology
SWEDEN
Abstract

Computing has made its way into most of our lives as a key processor of vast quantities of information. This has happened directly in terms of gadgets and devices that assist us in everyday life, but also indirectly, through the critical infrastructures that enables these devices to function. A key issue with critical infrastructures such as transportation, communication, power-grids and finance, is increasingly circular interdependencies. Because of this issue, a disruption in either one can cascade and have a global effect on the others. To manage these complexities, we are depending on a number of monitoring systems that allow operators and other stakeholders to, within their respective expert domains, discover disruptions as early as possible and then take appropriate actions.

These monitoring systems are not without challenges of their own. In addition to having evolved organically alongside their respective infrastructures, there is a considerable legacy to account for, with both hardware and software components spanning decades of computing history. This puts heavy restrictions on the kinds of interventions that can be performed safely, implying that these systems are ill fit for handling the software and software security landscapes of today, where updates and adjustments need to be applied on a daily basis in order to stand a fighting chance.

The work presented herein addresses some of the major challenges in securing these monitoring systems against current and future threats posed by antagonistic actors, dormant software defects and changes imposed by technological advances and academic discoveries. This is approached on several fronts in parallel: by embedding resilience in order to allow for controlled experimentation and evaluation of new protection mechanisms in incrementally sensitive settings; by developing laboratory facilities for resilient smart power-grids; and by developing tools and training scenarios for operators of adaptive and reconfigurable monitoring systems.
Acknowledgements

This work has been partially financed by the EC grants:
FP6-038576, INTEGRAL
FP7-238868, SEESGEN-ICT

Furthermore, I’d like to extend my sincere gratitude:

... To my “Pragmatiske” supervisor Lars Lundberg.
... To my “Ständige” supervisor Rune Gustavsson.
... To my “Resiliente” supervisor Per Mellstrand.
... To Nancy R. Mead for detailed and valuable feedback.
... To Jan Johansson, the MiB and the unsung heroes of “MiB-Cigar”.
... To Raphael Caïre, Nouredine Hadjsaid, Seddik Bacha, Bertrand Raison,
and all the other friendly figures at IDEA/G2Elab.
... To Johan Zackrisson, for providing much needed laboratory back-end work.
... To my beloved family, friends and relatives.
... To the formative Ghost of Christmas Past that was SoCLab.
... To the therapeutic, balanced and objective think-tank that was Lunchmobben.
... To all my fellow travelers on this arduous journey; those that happily tagged
along; those that I regrettably lost somewhere along the road; and those that inten-
tionally stood in the way.

Your contributions will not soon be forgotten.
# Contents

## I Introduction

1 Overview
   1.1 Introduction ................................ 3
   1.2 Organization ............................... 5

2 Background
   2.1 Information Systems and Information Processing Systems . . . . . . . . . . 7
   2.2 Resilience and Virtualization .................................. 13
   2.3 Software and Software-Intensive Systems .......................... 18
   2.4 The Origin of Anomalies ........................................ 21

3 Structure
   3.1 Mission Statement .......................................... 25
   3.2 Related Work ................................................ 26
   3.3 Hypotheses ................................................... 28
   3.4 Contributions ................................................. 31

## II Contributions

4 Self-Healing & Resilient Critical Information Infrastructures 39
   4.1 Introduction ................................................. 39
   4.2 Setting the Scene ............................................ 41
   4.3 Understanding and controlling complex systems ................... 43
   4.4 Experiment Environments ...................................... 47
   4.5 Other approaches .............................................. 48
   4.6 Conclusions ................................................... 49

5 Use and Misuse of Virtualization 51
   5.1 Setting the Scene ............................................. 51
   5.2 Approaching Virtualization .................................... 52
   5.3 Possibilities ................................................. 57
   5.4 Caveats ....................................................... 59
   5.5 Moving Forward ............................................... 68
   5.6 Concluding Remarks .......................................... 75

6 Experimenting with Infrastructures 77
   6.1 Background ................................................... 77
   6.2 Experimenting with Power Grids ................................. 78
## List of Figures

2.1 Abstract processor ........................................ 8
2.2 Systems of systems ........................................ 9
2.3 IPS Environment .......................................... 10
2.4 RAID vs Direct Storage ................................... 15
2.5 Von Neumann Architecture ............................... 16
2.6 A simplified SCADA Model ............................... 21
2.7 Two-tier bug taxonomy .................................... 22
3.1 Main Contributions ........................................ 31
4.1 INTEGRAL Field Tests Architecture ..................... 41
4.2 IICS Architecture ......................................... 44
4.3 NSF GENI Architecture .................................. 48
5.1 Hypervisor model ......................................... 53
5.2 Von Neumann Virtualizable Resources .................. 54
5.3 Machine-space vs Virtual-space ......................... 55
5.4 Virtualization Ideal ....................................... 56
5.5 The virtualization Problem ............................... 60
5.6 Dynamic Loading ........................................... 62
5.7 Scripted dynamic loading ................................. 62
5.8 Machine Feature Sets ..................................... 64
5.9 Possible actions ............................................ 68
5.10 Iterative process ......................................... 69
5.11 Non-native Representations .............................. 70
5.12 Virtualization Hierarchies ............................... 70
5.13 Reinforce Borders ........................................ 72
6.1 NSF GENI Architecture .................................. 78
6.2 Microgrid Model .......................................... 79
6.3 ICT overview ............................................... 80
6.4 The Virtualization Problem ............................... 82
6.5 EXP-I Borders ............................................. 83
6.6 EXP-II Services .......................................... 83
6.7 Lab Setup .................................................. 84
6.8 Lab Instance ............................................... 85
6.9 Unified Operations ........................................ 87
7.1 Tools and Organizational Units ......................... 94
This section covers strictly academic publications and their connection to Part II of the thesis (where applicable). For a more complete list of contributions, please refer to Chap. 3.4.

*Self-healing and Resilient Critical Infrastructures* [1], Chapter 4, Rune Gustavsson and Björn Ståhl, CRITIS2008, 3rd International Workshop on Critical Information Infrastructures Security.

*Analyzing Systemic Information Infrastructure Malfunction* [2], Per Mellstrand and Björn Ståhl, CRIS2009, Fourth International CRIS conference on Critical Infrastructures.

*The Empowered User - The Critical Interface to Critical Infrastructures* [3], Rune Gustavsson and Björn Ståhl, CRIS2010, Fifth International CRIS conference on Critical Infrastructure.

*Experimenting with Infrastructures* [4], Chapter 6, Ståhl, B.; Luong Le Thanh; Caire, R.; Gustavsson, R., CRIS2010, Fifth International CRIS conference on Critical Infrastructure.


Part I

Introduction
1 Overview

1.1 Introduction

At the heart of critical infrastructures are power grids. In the broader sense, power grids refer to the generation, transmission and distribution of electrical power and without access to electricity, most other infrastructures would simply cease to function in any meaningful capacity. Power grids are exposed to a wide variety of disturbances where causes range from natural forces (e.g. extreme or fluctuating weather conditions), to structural weaknesses stemming from regular wear and tear, poor maintenance, outdated configuration of protective devices all the way up to plain old human error. To better detect and respond to these disturbances, large monitoring and control systems (commonly referred to as SCADA) have co-evolved with the power grids to the point that the infrastructure and the monitoring systems are virtually inseparable.

These monitoring systems do not come without challenges of their own. Given their computing nature, there is an explicit dependency on computers and software and an implicit dependency on electricity, thus forming a circular dependency. In addition, computers and software are components that, compared to copper wire and power converters, have short shelf lives and are developed on a best effort basis where whatever represents the current 'best effort' is a rapidly moving target. Since these systems are necessary in the upkeep of the infrastructure, they are considered as having an online value. This implies that they cannot easily be brought down for replacements or upgrades without increasing the risk of contributing to disturbances.

While minor disturbances in power grids are a common and manageable part of day to day operations, these can introduce unexpected interactions that lead to larger disturbances and so on, ultimately cascading to what is commonly referred to as a blackout. Blackouts are massive power outages affecting millions of people with tremendous societal consequences. The July 2012 blackout in India [100], thus far the largest power outage in history, is a prime such example, with some 670 million people directly affected.

1.1.1 The Quest for Smarter Grids

To protect monitoring systems, the design philosophy has been to keep them separated from the outside world, e.g. the comparatively more exposed corporate or public communication network environments. This has had the unfortunate side-effect that the secluded SCADA environment is quite literally decades behind in terms of deployment of advancements in software and network security related
1. Overview

protection mechanisms. In addition to this, numerous real world examples (as will be shown in Chap. 8) illustrates repeatedly that this notion of air gapped security has been an illusion from the very beginning.

To protect the power grids themselves, there is an ongoing quest for ‘smarter’ grids. Smarter, in this context, has a wide meaning ranging from merely increasing the precision and frequency of monitoring samples to the restructuring of entire grids into a cell-like structure. Among the supposed benefits to the latter approach is to be able to take much better advantage of distributed and renewable energy resources (DER/RES) than what is currently possible, but also to open up for more flexible energy management schemes that are less susceptible, if not immune, to cascading effects from major disturbances.

Most visions for smarter energy grids depend on the successful deployment of Smart Meters, or rather, the next evolutionary step of preexisting Advanced Metering Infrastructures (AMIs) with higher sampling resolution and a higher sampling rate, but also options for empowering the end-user through interfaces for defining local energy management policies based on pricing, availability and the users own priorities.

1.1.2 Infrastructure Affordances

An affordance in the spirit of [6] is about the perceived action possibilities of an object and is more commonly used in user-centric design efforts, with a trivial example being that a chair has the affordance of sitting. With infrastructure affordances however, we refer to the complete set of possibilities that an infrastructure offers us. The subset of these that are accessible and usable by a certain stakeholder may vary with training, opportunity and domain of expertise. The affordances that a stakeholder subsequently depends on becomes his or her requirements.

The role of infrastructures is thus to provide affordances that match the requirements of its stakeholders. In matching affordances with requirements we need tools supporting adjustments and configurations of components (smart metering ecosystems) as well as tools that support monitoring associated behaviors and performance indicators. Thus, it falls on the hands of individual stakeholders to monitor this connection for changes that would bring local (to the stakeholder) adverse effects, as a fixed global set of requirements to standardize on or work from does not exist.

This brings us to the title of the thesis, Monitoring Infrastructure Affordances. By combining the notion of infrastructure affordances with the role of monitoring in future smart grids, we explore the idea that we can, by taking advantage of certain properties of how software behaves, build monitoring systems that allow individual stakeholders to both define and refine individual monitoring models in a safe and secure way. The focus is therefore on the overlaps between traditional monitoring roles (SCADA), future monitoring systems (Smart Metering Ecosystems) and their computing parallel, Systemic Debugging (Chap. 7).
1.2 Organization

The thesis is divided into three distinct parts: Introduction, Contributions and Experiences.

The Introduction part contains two chapters, Context and Structure.

The Context chapter (pp. 7-23) covers informal descriptions of a majority of key system- and problem- decompositions intended primarily as an aid for the Experiences part of the thesis to be accessible to a wider audience, rather than just to domain experts.

The Structure chapter (pp. 25-35) defines key challenges to be addressed, corresponding hypotheses subdivided into research questions and correlated to individual contributions and their respective forms of dissemination. Related work in the sense of projects and efforts that act as strict influence or even bias, is also covered.

The Contributions part is mainly a collection of previously published works, although in contrast to a strict compilation of published articles, these works have been both edited to better fit the flow and style of the thesis and been extended with better descriptions and more detail. Thus, these chapters are, in particular, written in such a way that they can be understood independently of the others, and may therefore – to the meticulous reader at least – appear as having slight redundancies or overlaps.

The Experiences part contains two chapters, Results and Discussion.

The Results chapter (pp. 123-140) begins with an Executive Summary (pp. 123-124) that presents a brief, high-level view of the problem domain, approach, key contributions and results. This is followed by an overview (pp. 124-125) of how the hypotheses from chapter 3 relate to the remaining three sections, Restructuring SCADA (pp. 125-131) ICT for Smarter Grids (pp. 131-135) and Framework for Systemic Debugging (pp. 135-140). These three sections combine selected results from Part II with the hypotheses and research questions from Chap. 3, but also extends them with additional results and lessons learned.

Lastly, the Discussion chapter (pp. 141-147) takes a critical and reflective stance to the work that has been done, contrasting it with related work in the sense of different, conflicting or otherwise competing efforts, rounding it all of with possible uses for the results in future industrial- and research- applications.

1.2.1 Reading Guidelines

To assist the reader in navigating the thesis, a few reading paths will be suggested.

For a brief overview, there is the executive summary (pp. 123-124) and the associated discussion (pp. 141-147).

For an in-depth technical coverage from a computing perspective, there is the chapter on Virtualization (pp. 51-75), Experiment Environments (pp. 77-90) and Tools (pp. 91-106), combined with the supplementary book, Systemic Software Debugging available through [70].
1. **Overview**

For a higher level *method* coverage from a *smart grid* perspective, there is the chapter on *Resilient Critical Infrastructures* (pp. 39-49) as well as the chapter on the use of *Intelligent Agents* (pp. 107-120) along with skimming over the core portions of the *Results* chapter (pp. 124-125).

For evaluating the academic merit of the work involved, there is the *Mission Statement, Hypotheses* and *Research questions* (pp. 25-26, 28-31), with corresponding *Validation* from (pp. 124-140), specific contributions (pp. 31-35, Table. 3.1 and Figure. 3.1 as reference) and *Related Work* (pp. 26-28, 141-144).

Some key terminology used is defined in pp. 149, but the reader is expected to be comfortable with software engineering terminology as per ISO/IEC/IEEE 24765:2010 [89] and secondarily through [7]. Additional overview for reference can be found through Security Engineering as per [71].

In addition, if the contents of a certain chapter or section appear confusing, ill-defined or even incomprehensible, return to Chapter 2 (pp. 7-23). If that still does not help, blame the author.
2 Background

The purpose of this chapter, in contrast to the others, is not to describe work that has been done as much as it is to try and bridge some of the key concepts that the reader is expected to be familiar with in order for the main contributions from Part II and results from Part III to be understood. The reason for this is that the ideas and approaches presented stem from a wide assortment of disciplines and areas and may thus need clarification.

2.1 Information Systems and Information Processing Systems

It is often claimed that software has, amongst other things, behavior. Definition wise, behavior concerns the response an organism gives to a certain stimulation, but it is a far stretch to think of software as a living organism, even though there are some entertaining similarities. Therefore, we broaden behavior to include describing patterns on observable and measurable reactions, output, as a consequence to some stimulation, input. With software, we can usually find some logical block, be it a function or an entire program, and tinker with its respective input and configuration and then measure its output, or lack thereof, in search of regularities, patterns. The only time when this can be done is during execution. Thus, only software in its running state can be considered as having a behavior.

2.1.1 Information Processing Systems

If we treat software as simply a mix of instructions that regulate how ones and zeroes should be moved around, we will have a hard time trying to find properties such as meaning, purpose, intent or quality. To illustrate this, one may take a debugger and attach it to a randomly selected process on one’s computer, pause the execution of said process, and then disassemble the coming few machine code instructions. There is little doubt that these instructions describe what is about to happen inside the system in a very precise manner, but to try and extrapolate meaning from the instructions alone is a pretty futile endeavor; software without context is just an Information Processing System (IPS).

An IPS can take information coded in one form, perform some operations to change it around a bit, and spit it out in a different shape. This is accomplished with the help of a processor (Fig. 2.1). The role of the processor does not depend on a computer to do the job. At times, we do use humans to fill the role as processors. While the digital processor is able to process many orders of magnitude more information than say, a human counterpart would, they are still both perfectly able to process information. A key difference other than the obvious ones
2. BACKGROUND

![Diagram](image.png)

Figure 2.1: The abstract processor in an information processing system, where we only concern ourselves with the information streams (inputs, the interface, the processing and the outputs), not the specifics of the processor itself.

However, is the required precision of the instructions that describe the intended processing. A human for instance, is perfectly able to accept instructions that are fairly abstract, while the digital counterpart requires instructions that are exact. The benefits of one in regards to the other can be argued quite extensively, but having either one as part of your solution does in no way exclude the other from participating.

A short example to illustrate the presence of humans as integral but insufficient parts of an IPS, would be that of an emergency service. Through a communication channel, a connection is established between dispatch (operator) and the person in distress (user). The operator tries to extract a set of relevant parameters to determine the gravity of the situation, decide what the correct response is (formulate a request) and then forward this request (dispatch) to some other part of the system such as an ambulance service, a police authority, firefighters or similar, depending on the nature of the dispatched request. The information used as a basis for this decision is weighted from several dynamic sources: the user, the pool of available resources, positional data from the communication channel and so on. The process is time-critical in the sense that the response time between an incident and the deployment of the most suitable remedial actions is key, but the elimination of the automated processing would cripple performance and the elimination of the human counterpart would render the service moot.

Now, it may seem strange to compare humans and computers in this way, but there are a few interesting points shared between the two in regards to the role as an information processor. One is how the information that is processed actually alters the future behavior of the processor. A mechanism that illustrates this is a cache, operating on the principle that information which is frequently accessed should be kept close by. This can be achieved in many ways: hash tables, machine-learning algorithms, metaprogramming reflection, dynamic recompilation, and many more. These techniques optimize the processing towards increasingly refined responses to information flows that are either identical or at least similar to the ones that have previously been handled, thus specializing the running program further towards information patterns that may well be specific to one particular setting, but which are also bound to the life-span of said instance unless accounted for. If the program is terminated, then it is quite likely that such dynamic fine-tuning will be lost. Specialization of this nature exemplifies what we elsewhere refer to as the online value of a system and is therefore substantially
different from other attributes such as its \textit{availability}. The merits of a system’s on-line value can also be thought of as information lost when rebooting or forcibly restarting a computer in the hope of reverting a malfunctioning system to a more stable state.

Optimization achieves two larger goals. The first one is to shorten the time that elapses from the input of a request to the output of a response. The second goal is that extraneous information, if any, can be omitted. A conceptually similar thing happens with the human processor when we gather experience. By doing something repeatedly you tend to get increasingly quicker and more precise as the number of iterations increase. The same basic benefits and problems persist however: processing gets tuned to a specific context, e.g. a work-place, and can be adversely affected if the subtle underpinnings of that context change or disappear, something which is likely to happen at least partly, when switching jobs or assignments.

The relationship between specialization and its opposite, generalization, in an IPS is particularly relevant when you have a system that is deployed in several, quite possibly hundreds of thousands of instances. Some system optimizations that are considered harmful in the long run, may need to be singled out and removed, while the good ones should be generalized to apply not only to the single instance but to all instances. Balancing specialization (optimization) and generalization is, by definition, problematic and one of the central challenges to engineering programs.

For instance, something that is often held as a virtue or strength of programs written using more abstract programming ”languages”, is that they can be detached from both the information that they process, and from the machine that enables the processing. It may well be a great thing that a program can run on several operating systems and in turn on several different kinds of processors and hardware architectures, achieving a far greater number of independently running instances, but that is still only made possible through yet another program which translates these more generic instructions into the specific ones that each single machine can act upon. To then be able to make good use of feedback like performance data, praise or anomaly reports, one instead needs to first filter irrelevant local traits from specific machines and then reinterpret this, hoping that the relevant bits were not lost in translation.

Figure 2.2: Systems of Systems, interconnected through interfaces. Viewed as a white-box (left) and as a black-box (right).
2. BACKGROUND

The next point of interest is how these systems can be expanded upon. That the output of one program can be the input of another is not particularly foreign; it is a core part of the imperative programming paradigm – but the central part is how these inputs and outputs are modeled and specified, the interface.

Provided that the interfaces of procedures, functions, methods, objects or programs make sense, i.e. provided they are defined in a way that one can hook into another, like pieces of a puzzle (Fig. 2.2), and have conforming information that should be propagated across these interfaces, a larger abstract picture can be painted: one of systems of systems. Ideally, any programmer capable of grasping an interface ought to be able to add to this growing machinery with ease. This is where black-box and white-box reasoning fits in; a programmer does not necessarily have to know the inner workings of the system(s) as such, he or she just needs to see to it that the interfaces match and that any rules on the sequence of information flows (protocol) are followed. During debugging or reverse-engineering, however, such a black-box approach may not suffice due to internal characteristics, such as the non-linear properties of the box. These properties include characteristics such as state sensitivity, feedback loops, recursions and concurrency. If you give a program the same inputs, but the outputs that you observe are different each time you run it, how can you expect to figure out its behavior, let alone establish if the observed behavior deviates from the intended or expected behavior, or understand the chain of events that caused the observed behavior?

![Figure 2.3: The enabling layer of any IPS, its environment.](image-url)

As it stands, the processor does not work on its own. There is a direct need for support from its immediate environment. As this model does not focus on technical detail, a lot of small subsystems are aggregated into the environment concept. Look at Fig. 2.3 for a quick illustration of some of the things that can be involved. Even with such a limited model, many of the properties that greatly contribute to making software problematic, can be explored.

This brings us to the final point of interest, which concerns the kinds of problems that may occur. The previous discussion on optimizations showed that optimization processes risk specializing execution too far; should any of the assumptions that a particular specialization relies on be invalidated from changes to the execution environment, the system may suffer performance degradation. Similarly, the discussion on interdependencies illustrated that systems can grow incrementally interconnected in subtle and unintended ways.
With such interdependencies, it is implied that the problems which affect one smaller part of the system can propagate, *cascade*, in essence making the entire system a brittle one. If you hit it hard enough at any point, the cracks travel far away from the point of impact.

Some information processing activities are definitely sensitive to timing in regards to both when processing begins and how long it takes. Reactive scenarios are particularly sensitive, like the emergency service example where the difference between the desired outcome and a catastrophic one relies on a proper response. A common problem here is the malfunctioning of a subsystem, be it a supporting technology like a hard drive or even in the computations because of program flaws. Such malfunctions typically force the subsystem into a terminal state from where execution halts, likely affecting the programs next in line.

From the processing part of information processing, it is easy to infer that this part implies some sort of change being imposed on some information stream; what the contained information represents or where it is currently being stored. Processing may only involve moving information from one location in storage to another, i.e. from one system to another, but could equally well involve quite extensive transformations, e.g. combining several pieces of information into a new one based on a delicate set of conditions. Chances are that the transformations deviate slightly from what was intended: a small negative number turned into a very large positive one, a real number losing a few digits of precision or a string of characters losing a few letters or perhaps shifted around a bit; whatever the reason, the effect is that information is corrupted. If such corruption is not discovered and dealt with in time, it too will propagate.

2.1.2 Information Systems

Thus far, we have briefly covered the purpose of IPSs, but the close associates of these systems, the blood stream of the computing world, the information system, is still to be discussed. As previously stated, merely processing information according to a dynamic and adaptive set of rules and regulations is a fascinating subject for study. However, there is usually some larger task or challenge\(^1\) that the processing is part of resolving. These tasks can be virtually anything from helping people to travel or communicate long-distance, to diagnosing a sick patient. In this thesis, such tasks will be referred to as *services*.

Many services can benefit greatly from automation, and technical innovations tend to have a large impact on what we spend our time doing. Leaving menial labor to machines allows us to do other, hopefully more interesting things. Many innovations on automation have in the past been of a mechanical nature: the spinning jenny, steam engines, and so on. What computers have done is to finally open up more abstract automation to a large number of people. While truly a great thing, this added layer to services depends on a very delicate machinery that we know to be flawed in many ways. We also know that correcting these flaws can be both difficult and time-consuming. Parts of this difficulty stems from

\(^{1}\)Something more than studying software simply for the pure joy of exploring the possibilities and structures of the abstract microcosm of computation.
how the various layers grow together to form a very complicated and dense blob. The distinct separation discussed here, between information processing systems and information systems is rare to find in society, and for good reasons. The following example illustrates this on a technical level:

A programmer writing a program for a very primitive machine finds himself in the need to store two bytes somewhere in memory. Picking an address at random is a hassle both for the programmer and for the machine, especially since there are a lot of subtle rules that regulate what can and what cannot be done at different locations in memory. For the sake of convenience, a small region of computer memory is reserved to be used as short-term storage, which we call the stack. A Central Processing Unit (CPU) register is reserved to keep track of where the base of the stack is located and instructions are included that add and remove items on the stack accordingly. The programmer can now focus less on getting memory access exactly right, and is thus free to more easily make sure that his actions will not interfere with other parts of the program. Even if the program is flawed and ends up writing or reading from the wrong offset, the region to focus on is now quite small, neatly packed without a lot of random noise and with predictable access patterns to boot. The stack is an abstraction so primal that it is taken for granted. But using the stack, the programmer now needs to keep track of the position of his value in relation to the base, instead of the exact address in memory. However, even that becomes confusing after a while. To resolve this problem, the programming language can be improved to track symbolic references. This can be accomplished by the inclusion of a translation phase where symbolic references or ‘names’ are replaced by lower level counterparts. From these rather small changes, the task of instructing the machine how to store these two bytes has changed:

1. **STORE** 0x22, 0x1020 - storing directly.
2. **PUSH** 0x22 - storing indirectly, relative to the current stack position.
3. **pumpValveStatusFlag = 0x22** - storing indirectly, using a symbolic reference.

Now, with the third option, two major things have happened. One is that the instruction is more abstract – the addressing done could in fact be either as in 1 or as in 2 but what actually happens will depend on the tool that performs the translation. The other thing that happened is that small fragments of the meaning or intent behind the service and its respective information system have been encoded into the schematics of the processing.

This example hints at something which really represents a large shift in what both programming languages and programming are all about. We are actively piling layers upon layers of abstractions, and expanding the set of tools needed to translate these abstractions into the basic building blocks that actually do something. These layers enable the coupling of the processing to the information system.
The above example only covers the trivial use of a variable, but a good exercise is to try and think about what must necessarily be done to make current ‘cutting edge’ programming languages work in terms of object encapsulation, inheritance, polymorphism, generics, reflection and so on.

From an analysis standpoint, it would be simple to say that what all this boils down to is that we can reason backwards from an observation of something undesired and trace it back to whatever flawed statement that caused the behavior in the first place. With that out of the way, the only thing left is to figure out what might be wrong with the statement and finally change it into something less flawed. This may well be the case for extremely trivial problems. The more complicated ones, however, can rarely be explained away by something trivial, such as the use of an uninitialized variable. This leaves us at a point where we are forced to consider the chain in its entirety: from the service, to the information system, to the processing, to the processor. At times this mean that you need to be able to understand something fairly technical, e.g. CPU cache coherency. At other times it means figuring out what the client really needs and, perhaps more often, what some programmer actually meant with something like `pumpValveStatusFlag`.

### 2.2 Resilience and Virtualization

The ideal is, of course, to somehow create systems that are without any notable flaws. As far as the software part goes, this might be achieved by having requirement engineers encapsulate the goals of an intended software in the form of precise formal specifications which could then be dispatched to the other developers: system architects, designers, programmers, testers and others. They, in turn, would sit down and with perfect rigor first develop the system, then deductively prove its correctness, and finally deploy the solution to a presumably satisfied customer. In other words: one solution seems to be to elevate the art of software development to a principled engineering discipline. Developing software ought to be no different than constructing a bridge, a spacecraft or a new car.

Unfortunately, such comparisons are at the very best poor and misguided. Software – in the sense of the combination of code and data that through the execution by a computer enable computation – is not even remotely similar to a bridge, a chemical, a car or a quilt. In fact, the very characteristics that make software so versatile, also constitute the reason why software is difficult to capture and control using traditional mathematical models and methods.

These characteristics spring from, in part, the fact that interactions are heavily dependent on feed-back loops which introduce non-linearities when they are finally executed. Other contributing factors are the strong interconnectedness between the software, the machine that makes the software tick and their respective environments. In this way, software is controlled to a great extent by the actual limitations of the machine in terms of computational capacity, storage space and communicational bandwidth.
2. BACKGROUND

In addition to this, system components are often executed in environments that was not, and could not, be fully known at the time they were designed and developed.

An alternative to the ideal of flawless software just described, is to, with a systemic perspective in mind, embed resilience. This means that systems are designed and developed with the notion that no component is ever to be assumed flawless and that any component present will decay with time. Obviously, a software component cannot be said to decay in a literal sense of the word, but as its environment changes the function of the component may be affected in strange and unwanted ways, invalidating previously strong assumptions with consequences that are hard or even impossible to predict. Things will eventually go awry, but if we accept this, we may be able to manage it to a certain degree. In short, to embed resilience we can:

- Decouple components, especially critical ones, in order to isolate problematic parts and limit cascading\(^2\) effects.
- Implement self-healing mechanisms, in order to recover from component failure.
- Recurrently strengthen the system through hardening fueled by continuously validating behavior – since self-healing and monitoring provide feedback on component stability and data on errors as well as on the respective impact of faults.
- Monitor component conditions and states in order to refine hardening and provide feedback for future development efforts.

These concise principles, primarily derived from [72] do not necessarily have to apply directly to software. Self-healing, for instance, is complicated to define and cover when just considering the context of software. To find a more relevant subject of study, it is more useful to work with several other systemic distinctions, the overarching one being that of software-intensive systems. These are systems where software is a necessary but not sufficient component. In such systems, the following attributes are especially emphasized;

Looking at one of the major weaknesses pointed out with IPSs, their supposed brittleness, it is easy to see how the two stances on development strategies that was brought up in the introductory chapter fit in. If it was possible to create a system that fits neatly into an extremely well-defined role with rigorous methods and deductive proofs, then there would be no brittleness to speak of. If this does not seem reasonable to achieve, and possible consequences of certain flaws are somewhere between unacceptable and disastrous, we are forced to try and lessen those consequences by patching wherever the seams fail. To combat the shifting quality of the various components that, when put together, make a modern program tick, developers turned to resilience.

\(^2\)Cascading is when the occurrence of a fault somewhere in a system propagates.
2.2. Resilience and Virtualization

The human body is a nice example of a very resilient system. This resilience is demonstrated by its ability to cope with some of the many problems that arguably stem from a combination of its schematics – the genetic code – and contact with the pathogens and contaminants that are always present in its surroundings. These problems can be presented in a conceptually similar way to what goes on in an IPS, i.e. systemic effects may be brought on as a consequence of a damaged component or from unforeseen interactions between components. However, the body is in many cases able to repair and rewire itself and therefore cope with a large assortment of injuries that would otherwise be fatal, i.e. self-healing. While computer systems may not yet be able to repair themselves to the extent that humans can, there are a number of interesting technical examples which illustrate some degree of resilience:

**RAID** – Hard disk drives have long been the source of much headache. Not only are they quite central to the operation of most personal computers, they often contain both sensitive and valuable information. They are quite easily replaced, but when they fail their content is usually partially damaged or even unrecoverable. As they consist of many mechanical parts that are put under heavy strain, and at the same time sensitive to both shock and temperature, their lifespan is quite short. Several strategies have been developed to deal with the consequences of data-loss due to a failing hard-drive such as regular snapshots (backup) and redundancy. One particularly interesting idea in this regard is Redundant Array of Inexpensive Disks (RAID) [8]. This system comes in several flavors, but the typical pattern is that two or more physical disk drives are combined into one logical *volume* and by distributing data across these drives in intelligent ways, various benefits are achieved.

![Diagram of direct access compared to RAID-3](image)

Figure 2.4: Illustration of direct access (top) compared to RAID-3 (bottom).

Some of the variants of RAID use a parity component which means that a certain proportion of storage space is reserved for storing parity information (error-correction codes), providing the benefit that some or all information, that would otherwise be lost due to a damaged drive in the array, can be recreated.

1. **Storage capacity** – Because some space is used for storing the parity data, the amount of available storage space is considerably less than the total capacity of the drives in the array.
2. **Performance** – For every write to the virtual volume, the parity must be calculated. To do that the data must be routed through a processor, either on a shared CPU or on a dedicated one. This alters the load on different buses and possibly makes some optimizations such as Direct Memory Access (DMA) transfers less efficient or useless altogether.

3. **Virtualization** – The operating system must ensure that no device driver and no part of the Application Programming Interface (API) allow the virtualization to be circumvented by raw device access.

4. **Coordination** – To implement the solution, several other tasks may need to be coordinated and taken into account. This includes the scheduling of read/write operations and individual drive states, such as head position and rotation speed.

The overhead, including the maintenance of additional software/hardware that performs the actual translation, can be considerable and this is not atypical of resilience mechanisms. While the benefits of RAID may not always warrant the overhead, there is another technology whose merits are seldom disputed: process separation.

![Figure 2.5: The Von Neumann architecture of a stored-program machine.](image)

**Processes Separation** - An extremely potent resilience mechanism is process separation, a kind of virtualization now present in most operating system kernels. Its distinct benefits are most easily illustrated by looking at what happens when it is absent.

Start by winding the clock back a few years and assume that we have a very limited Von Neumann machine, Fig. 2.5. This means that program code is stored in read/write random-access memory from which a processor with flow-control and arithmetic support fetches instructions based on the value of a designated register, interacting with its immediate environment through inputs and outputs. Note that there is no direct separation between code and data. Locations in memory can contain data which can represent either one, or even be both at the same time.

Consider the situation where we need to run many separate bounded computations (programs) at essentially the same time, while limited to using a single machine. This is the very basic idea behind multitasking.

There are two fundamentally different ways of implementing multitasking: cooperative and preemptive.
Cooperative or non-preemptive means that you write programs in such a way that they cooperate by turning the control of the execution flow over (called yielding) to each other whenever feasible. Typically, this is implemented using an agreed upon memory region structured as a queue, array, or list of pointers to the current instructions of all participating programs. Yielding execution thus means adding your next instruction to the structure and jumping to the address of the next logical entry in the structure as often as necessary or feasible (safe).

When using preemptive multitasking, the programmer (or the tools if such a feature is part of the run-time support required by the programming language used) does not, in principle, have to make any special considerations as a smaller program, the scheduler, is given access to an external trigger that will forcibly halt execution of the current program and transfer it to the scheduler. A common such trigger is an interrupt handler connected to a hardware timer.

Whichever way it is implemented, multitasking has both strong benefits and suffers from considerable problems. One such problem is that the set of programs active needs to both be stored in and have access to memory. With multiple programs sharing the same memory space, careful consideration needs to be taken to avoid a situation where the memory addresses used for storing data and instructions for one program do not overlap or collide with others. Provided that the memory space is sufficiently large to fulfill the needs of both programs, this would be possible to calculate during construction, had it not been for indirect memory operations.\(^3\) The big issue concerns what happens to the rest of the system when a part of the program or hardware misbehaves. A very common class of bugs concerns a write to an unintended location in memory which may then alter the data or instructions belonging to another program. This may change the outcome and behavior of that program, in ways that are extremely hard to predict. Such a change is likely to cascade, ultimately rendering the entire system unstable and eventually broken. Reverting the system to a working state then requires a restart, a very time-consuming operation and critical data belonging to otherwise working programs may be lost.

Process separation can lessen this effect quite dramatically. By modifying the scheduler to only keep one program and its respective data in active memory, temporarily moving everything that belongs to other programs to a safe place, the ability of one program to malfunction and cascade into another is significantly reduced. At the same time, the illusion of several programs running in parallel is maintained. One way to implement this is simply to use the input/output capabilities of the machine and appoint a hard drive or similar device for swapping program memory contents back and forth. However, this interacts poorly with the communication bus used for connecting the processor with the swap device and thus becomes a last resort unsuitable for highly frequent task switches. A more clever way to deal with this is to virtualize memory by adding an intermediate translation step to each and every memory access.

---

\(^3\)Indirectly, this means that the actual address is the result of a previously computation, using data from other memory locations or CPU registers.
This translation maps a path between a virtual address and a physical one using a reprogrammable lookup-table. This allows for a number of interesting possibilities, like extending the amount of accessible memory beyond the size of the address bus\textsuperscript{4}.

This actually covers a lot of the microcosmos within operating systems. If the individual tasks here are refined extensively, something like the modern operating system ought to eventually emerge. At the same time, this discussion also illustrates a few important things, like the tendency to reshape one problem (here, \textit{cascading data corruption}) into another one (\textit{performance degradation}) while at the same time increasing overall complexity, by employing an absolutely central concept, \textit{virtualization}. Paradoxically, the solution to the degraded performance is then to embed (or hide) the implementation in the machine, thus increasing the complexity of the hardware involved.

In closing, the resilience mechanisms illustrated are quite potent but also inherently dangerous. In many cases they are simply not necessary (the system continues to behave as expected). If they are activated and actually perform their set task, the underlying trigger needs to be examined and dealt with, i.e. debugged. Hence, resilience mechanisms need to be actively evaluated and monitored. The means for how this can be achieved in the context of virtualization as a possible resilience mechanism is the subject of Chapter 5.

2.3 Software and Software-Intensive Systems

\textit{Punch cards} as a way of storing information predates modern computers by a long period. They were initially used for storing instructions that controlled mechanical automation such as machines used for weaving fabric or self-playing pianos. With the advent of early computers, punch cards were both an accepted and useful form of storing instructions as the alternatives were expensive and cumbersome. Discarding punch cards as the means for storing instructions and data, we can assert that the focus of early software was the automation of calculation of things such as the numerical approximation of non-linear differential equations and other tasks of a similar nature. With \textit{punch-card software}, we figuratively mean the idea of software created for a limited and well-definable purpose with few or no dynamic properties. Such software was not likely to change or be adversely affected by smaller changes in its immediate environment. This marks the border between what used to count as computers and what is now generally meant by the term.

The next step follows a very predictable path: the technologies behind computers developed rapidly and advanced exponentially in terms of capacity, ability and availability. Closely trailing the tracks of such advances were comparable changes in the programming languages used to generate the code that coordinated and controlled the different technologies. What the programmer focused on switched from the implementation of a single algorithm to stitching several

\textsuperscript{4}For a taste on how this used to be done, read up on a technique called \textit{bank switching}. Although somewhat outdated, the underlying principles are still relevant, with very interesting and far reaching consequences.
algorithms together using a sort of data-structural glue. This mindset was conveniently called structured programming and object oriented programming can be considered an advanced form of this type of programming. Along the same lines, computing got applied to new key areas of application where the management of information was the new priority. Secondary storage of programs switched from punch cards to Read Only Memory (ROM) chips, hard-drives and other forms of persistent storage. Programs went from being one-shot calculations to tools for managing, analyzing and manipulating large quantities of information.\(^5\)

With almost each and every advancement, new layers of abstraction were introduced and piled on top of previous ones in order to try and manage the new demands. Communication networks brought on all sorts of major changes, particularly large-scale collaboration between computers. To speed things along we can skip a few steps and simply state that software has slowly changed from basically sorting strings and factoring prime numbers into something highly dynamic, capable of operating both distributed and locally while at the same time reshaping and tuning itself to current operating conditions and usage pattern. In stark contrast to the mid twentieth-century market for computers, which has notably been estimated to suggest a total demand of approximately five to fifty computers world-wide, we have since long passed the point where computers, more potent than previously imaginable, outnumber us by far.

Most of these computers perform fairly simple tasks and consist of digital probes for temperature and humidity, alarm clocks, etc. However, if we take a walk, actual or imaginary, through a store in a supermarket chain and look for the information systems which they govern, we will notice them everywhere. They are found in the Radio Frequency Identifier (RFID) or in the barcode enabled hand scanner that you run over each and every item before you add them to your shopping basket. The little key that the scanner reads connects data stored in a local database that acts as a cache for the chain-wide super servers which keep tabs on people, accounts, prices and possible discounts, continuously updated using data from stock markets and news agencies. These databases are linked together over the corporate virtual private network, layered on top of the internet. This goes on for quite a while, but the core matter is that a stunning amount of day to day activities is heavily regulated and managed by computers, interconnected and working in ways that are hard to cognitively grasp. It is neither accurate nor particularly fair to reduce it all to being hardware that performs calculations based on a static set of rules. The main keywords here are thus open, heterogeneous and dynamic:

- **Open** because the borders are undefined. They really depend on where you draw the line based on your roles and stakes in the system. It is tremendously difficult to measure and find strong interfaces where you can comfortably say that this is the point where your particular system, responsibilities and problems begin or end.

\(^5\)For a slightly more academic touch on the subject, please refer to the great principles of computing, http://greatprinciples.org
• **Heterogeneous** because few components are ever likely to be similar and interchangeable. Lots of processors, varying from newer 64-bit Reduced Instruction Set Computer (RISC)/Complex Instruction Set Computer (CISC) hybrids with reprogrammable instruction sets sporting features such as hierarchical privilege separation, to hardwired 8-bit ancient technology still hidden away inside cash registers and similar devices, may happily cooperate over a shared communication bus until the day one of them starts acting weird following some corporate enforced policy on date and price representation change.

• **Dynamic** as the total set of active components and their respective connections change quite frequently, often on several levels at the same time. Policy changes, political reform and technological advances all the way down to adaptive optimizations and normal wear and tear, all contribute to the dynamic behavior of the system.

With this in mind, the previously stated idea of software as merely automated calculation is clearly inadequate. Communication, information, interaction and other advanced properties are considerably more relevant. Systems overlap and share components on several levels, with the more obvious ones being things like dynamic libraries, communication stacks and file-systems in multitasking operating systems. All components are susceptible to some sort of life-cycle; they age, get patched, re-tuned or replaced on a somewhat regular basis. Different components tend to age at different rates, however, and even though some seem to interact fine at first or even most of the time, subtle incompatibilities can suddenly be introduced.

### 2.3.1 Infrastructure Monitoring: SCADA

As an example of a software-intensive system rich in legacy, consider an instance of a System Control and Data Acquisition (SCADA), which is basically a telemetry and telecommand successor for the management of production facilities, power grids, flight control, surveillance systems, etc. A pioneering application for the first half of the 20th century was support to weather forecasting where computational tasks were performed by humans and results and measurements were conveyed through the use of telegraphs and radio. This was refined and optimized with technology from other industries such as railway management that had similar problems with gathering measurements. Humans were eventually replaced by sensors and relays communicating by radio. In the early 1960s digital computers had advanced enough to be integrated.

At this point the SCADA model (Fig. 2.6) emerged, with Remote Terminal Units (RTUs) gathering measurements and enforcing commands. Each RTU communicates with a Main Terminal Unit (MTU) that aggregates and records data but also interacts with a Human Machine Interface (HMI).

The HMI is used by one or several operators that manage whatever resources the system controls. Although technology has shifted considerably since then, the basic idea and layout remain the same, and the biggest change in technology related
2.4. The Origin of Anomalies

to SCADA has happened in surrounding systems; corporate Information Technology (IT) became, for better or worse, ubiquitous. Digitalization became the new word of the day, and everything from customer records to incident reports and technical schematics were stored in databases and made accessible through the local Intranet at a response time of mere milliseconds. To improve operator decision making, information from other parts of the corporation should be accessible to the operators working at the HMI. Eventually, the comparably frail SCADA system was bridged with other networks.

Now, well-established SCADA systems in long-serving infrastructures have, as just mentioned, a considerable legacy. Samples from most parts of computing history can literally be found still running at some well-forgotten remote terminals. The next step would be exposing subsets of the information gathered to the customer base as to reduce loads on customer support and similar benefits, like a power grid operator with a website showing current network health.

Now, think of the complexities dormant here: old computers well beyond retirement are coupled with their cutting-edge counterparts, both with software of their own written in languages both prominent and expired, communicating using a mix of poorly documented proprietary binary protocols and standardized open ones. In addition, these are sharing networks with other systems within the corporation: accounting, storage systems, the odd virus and so on with pathways at times leading as far as to the Internet.

2.4 The Origin of Anomalies

Our approach to resilience is twofold. One is to improve the actual composition of the system, by any means possible. The other is to have an organizational structure that is capable of taking care of issues that arise in an expeditious but careful manner. Probably both the composition and the efficiency of the surrounding organization will need to improve the end service. The concern shared by both approaches is the range of problems that can occur 2.7 as per Von Neumann 5.2, and the common causes that precede them.

The principal claim on the supposed origin of software anomalies that is often discussed in software-engineering literature [73] is that each and every anomaly is simply an inconsideration on behalf of the developer, but this may just be one of
the cases where things are being made a bit too simple due to convenience rather than brevity; in the world of a developer there are literally hundreds of protocols, conventions and interfaces within the machine, language and execution environment that, to a variety of degrees, need to be followed just to be able to create, by current standards, pretty trivial software. Some of these interfaces and protocols are poorly defined, change as the project progress, lacking either representative documentation or being outright ambiguous, sometimes by design. Factor in other related concerns such as security, developing tests, proofs alongside wiggle room for future alterations into the mix and we are looking at a pretty complicated mess. In addition, there’s a blossoming category of tools that purposefully exploit ambiguities and convention to obscure and obfuscate the actual workings of a particular piece of software [74].

In order to be able to cope with this complexity, tools for automation have steadily increased. Few today are capable of manually creating machine code, or linking together compiled code into an executable binary or even supplying the compiler with the optimal set of parameters. In spite of such tools, the situation in regard to complexity has not reduced the challenge that developers face. On the contrary, the major change is that focus has shifted from technical detail to other parts of the process and other levels of abstraction. Consequences from an inconsideration, however, are at least as grave as before; they are amplified by an increasing divide between description (code), transformation (toolchain) and execution (machine/environment).

An early example of the dangers of overextended trust in the tools we use to construct software can be found in [9] wherein Thompson describes how he exploited the learning facilities of a C compiler capable of compiling itself to have it output different (with a backdoor) code when and if the Uniplexed Information and Computing System (UNIX) login program was compiled. The toolchain of today is far more convoluted; we have compilers, linkers, virtual machines, code signing, code encryption, code obfuscators, CPU microcode, virtualization extensions, copy protection, runtime packers and unpackers, etc.; the list goes on for quite a while. Which ones of these can be trusted to not introduce subtle and hard-to-detect problems?

![Figure 2.7: Two-tiered minimalistic taxonomy of observable effects and corresponding labels.](image)

A second claim on the origin of anomalies concerns the way we make assumptions about the inner workings of some particular system and how we thereafter proceed to make alterations for our own benefit by either changing the original source code
or by exploiting some interoperability/modularity features (such as dynamically loadable libraries). While any such alterations may work fine on their own when the subsystem in which they operate is considered, we may have inadvertently changed the situation for some other party that we indirectly share or compete with for some resource. If such a change is not taken into account by other systems, we, through feedback loops, inadvertently worsen our own situation.

Not only are these situations prone to anomalies but the potential problems that may arise from such situations are complicated with behavior that may depend on environmental factors in the enabling machinery sometimes differing radically between each instantiated system. This happens both on a small scale with third-party libraries or, as is the case with many modern development environments, the built-in API where we assume and rely on tools and feedback from execution (like iteratively alternating between code completion/code insight for API parameters and a test run to see if the immediate behavior was as intended). This also occurs at a large scale from the integration of components and services when implementing, for instance, corporate wide information systems.

The third claim is the *changing environment*, which relates to the fact that surrounding systems we depend on for our own systems to function properly change in some way not accepted or accounted for. This happens at an escalating rate due to the high availability of communication channels between a system and its creators/maintainers through automated software updates over the Internet. The properties and components of a large software system may suddenly change radically in just a short period of time without much further notice, and while many development efforts strive to maintain compatibility to as large an extent as possible, this often becomes a task even more complicated than developing the software in the first place.

These three claims are fairly similar. We assume things and fail to consider something complicated; the surrounding changes, and all of a sudden our software stops performing as intended or expected. Unfortunately, none of these *anomaly origins* seem especially preventable; they merely relate to the many unavoidable challenges in the manual translation between an informal description and a formal one – the actual task of programming.
3 Structure

In this chapter, the fundamental issues that we work towards resolving are first presented in the context of a Mission Statement. This is followed by a brief section on Related Work that covers academic and industrial efforts that, to some degree, act as influence or bias. Then, the third section, Hypotheses, lay out the principal propositions towards addressing the problem stated in the Mission Statement. These propositions are then subdivided into a series of research questions. Lastly, the specific contributions that stem from working on the research questions are presented, with particular emphasis on those that comprise Part II of the thesis and how they interrelate.

3.1 Mission Statement

Briefly put, the overall goal of the work behind this thesis is to enable the transition of brittle software-intensive infrastructures into resilient software-intensive infrastructures.

To clarify, the premise is that critical infrastructures, with a focus on the power-grid, are deemed critical due to their role in the upkeep of current living standards. There would be a substantial and significant loss suffered should such infrastructures fail. Whatever role an infrastructure fulfills, it has reached that position through gradual adaptation alongside other societal functions. As such, even though the initial designs might take scale and future development properly into account, the development on dependence of other infrastructures is unavoidable.

In the case of the power-grid, the benefits gained from telemetry and telecommand solutions – the predecessors to what eventually became SCADA class systems – outweighed the seemingly more distant disadvantages of being coupled to information and communication technology (ICT). As things currently stand, however, the dependence is circular, i.e. to maintain and operate the power-grid, a lot of Information and Communication Technologies (ICT) is needed. These ICTs requires a reliable energy supply.

Moving from bad to worse, developments during the last 15-20 years in regards to software security, have poked irreversible holes in the presumed security and barrier protections of these systems. There are numerous, previously improbable or implausible threats directed towards SCADA class systems and sophisticated, directed attacks have recently been revealed [101, 102].
In addition to this, indirect threats have been partially responsible for large incidents, such as the case with the BLASTER worm’s possible contributions to the much discussed northeastern blackout of 2003 in the USA [103].

The situation appears similar in other infrastructures that have become dependent on ICT, such as health care and transportation. Even though it may still be debatable whether the Internet is critical enough to be considered a critical infrastructure, the trend suggests that it will only be a short while before any such debate is laid to rest when the borders between smart phones, laptops and computers is further blurred as these technologies grow interconnected with current and coming services for identification, banking, healthcare and government. The problems facing critical infrastructures have not gone by unnoticed and there are numerous and strong ongoing efforts towards improving the current state of affairs. Some of these approaches directed towards energy supply management, e.g. micro-grids and smart-grids, aim at restructuring current solutions, and constructing new ones, into more robust and adaptive structures, and as part of this the associated SCADA systems undergo similar revision.

A major goal for future infrastructures is thus that they should be dependable and fault-tolerant, but also resilient. This means that they should be able to reconfigure themselves to harness disturbances. In fact, resilience is needed on several levels, both in the design and in the structure of the system itself, and also on a higher, organizational and a lower, technical level. This is not to imply that the undertaking of restructuring infrastructures is as simple as implementing the right mechanisms in the right places, there are also inherent contradictions and conflicts to consider. The Internet, for instance, is a prime example of a resilient structure – unsurprisingly as the ability to withstand losses of large portions of subnetworks was a major part of its initial design. Since the ability of data packets to be rerouted around failing nodes provides resilient and fault-tolerant communication at one level of the Open Systems Interconnection (OSI) model [90], the advantages of this function can be wholly undermined by the inclusion of more poorly designed protocols at higher levels. In any case, since it is likely that as the prospect of software-intensive systems will have an increased role in current and future infrastructures, essentially forming software-intensive infrastructures, it seems reasonably worthwhile to investigate the ways in which all the peculiar details of software can support or undermine infrastructure resilience.

Thus, among the principal resilience mechanisms found [75], the monitoring of dynamic behaviors regulated by virtualizations in the context of software-intensive critical infrastructure, due in part to the dynamic reconfiguration imposed by other resilience mechanisms, e.g. self-healing being susceptible to unforeseen interactions causing undesired consequences, resonance, was deemed to cover a sufficient set of problems and challenges with both academic and industrial merit, relevant to the involved stakeholders and other interested parties.

3.2 Related Work

There are, of course, several bodies of work that are either complementary, contradicting, similar or in other ways influential to the ideas that are presented in
3.2. Related Work

This thesis. The major works that were taken into account, acting as a strong influence or bias will be covered briefly. Other, more indirectly related work, can be found in Chap. 10.1. In terms of European research efforts, the projects directly related to this thesis are CRISP [104], INTEGRAL [105] and SEESGEN-ICT [106].

CRISP

distributed intelligence in CRitical Infrastructures for Sustainable Power had the cited goal of investigating, developing and testing how the latest advances in distributed intelligence by information and communication technologies could be exploited in novel ways for cost-effective, fine-grained and reliable monitoring, management and control of power networks with high degrees of Distributed Generation (DG) and Renewable Energy Sources (RES) penetration. The project was concluded in 2006 and, among the deliverables, D3.1 [107] and D5.3 [108] are related to the design and implementation of an experiment environment that was in fact the predecessor to the one described in Chapter 6.

INTEGRAL

Integrated ICT-platform based Distributed Control in Electricity Grids has the cited objective of building and demonstrating an industry-quality reference solution for Distributed Energy Resources (DER) aggregation-level control and coordination, based on commonly available ICT components, standards, and platforms. Furthermore, the project aims to demonstrate how this is practically achievable within a short to medium time frame. As noted in the Acknowledgement section, the work underlying this thesis was partially funded by this project.

The practical validity of this project is to be shown through three field demonstrators, covering the full range of different operating conditions:

1. Normal operating conditions of DER/RES aggregations, showing their potential to reduce grid power imbalances, optimize local power and energy management, minimize cost etc.

2. Critical operating conditions of low voltage DER/RES aggregations, showing how DER can benefit stability when integrated with the main grid.

3. Emergency operating conditions showing the self-healing capabilities of the grid components.

Among these different demonstrators, the emergency operating conditions appeared to be the most relevant target while planning the work behind this thesis. This is because the demonstrator covered self-healing, time-critical components and a brittle software-intensive infrastructure with a large legacy. Chap. 4 covers the INTEGRAL project in more detail.

SEESGEN-ICT

Supporting Energy Efficiency in Smart GENeration grids through ICT – has the cited objective of producing a harmonized set of priorities to accelerate the introduction of ICT into the Smart Distributed Power Generation Grids and to investigate
3. Structure

associated requirements and barriers. (SEESGEN-ICT) aims to produce policy recommendations, identify best practices and draw scenarios and road maps for the next generation of electric distribution network.

As noted in the Acknowledgement section, the work underlying this thesis was partially funded by the SEESGEN-ICT project.

Among the results, parts of the deliverables D3.1, D3.2, D3.3 and D3.4 cover some of the discoveries made here.

3.3 Hypotheses

From the description in the mission statement (Sect. 3.1) we hypothesize that:

I. It is possible to restructure a preexisting static hierarchical monitoring system (SCADA) through principled virtualization and gain the ability to iteratively perform increasingly sensitive experiments on otherwise brittle subjects.

II. To counteract vulnerabilities imposed by dynamic reconfiguration through self-healing activities, existing monitoring facilities should be re-purposed to allow interleaving of different expert domains into combined monitoring models, in contrast to planar ones where different sets of stake-holders monitor only variables of immediate interest.

III. Dynamic, interleaved, monitoring models as suggested by (II), can be constructed through dynamic instrumentation of pre-existing software components without compromising response times in the monitored infrastructure.

3.3.1 Research Questions

The aforementioned hypotheses are subdivided into a number of research questions, which are defined as follows:

RQ1– Which principal mechanisms exist for enabling and improving resilience in software-intensive systems?

Software can have varying degrees of resilience. These degrees depend in part on the structured control of abstractions that model the execution that the software’s code describes, but they are also in part based on the design and construction of the components which, when put together, comprise the software’s environment. The connection towards the latter can be readily illustrated by comparing the consequence of a malfunctioning software that fails to adhere to any of the critical, enforced protocols which define the boundaries of the interaction between the software and its environment, first in a operating system model that do not enforce a process based separation of privileges, such as MS-DOS, with ones that do, e.g. most UNIX kernels. Although the separation itself does not protect the executing process from any of the terminal conditions (access of unallocated memory pages, illegal instructions and others) other processes running, and, of course, the operating system kernel itself, will be able to continue. Looking at related, low-level structures, we can discern similar patterns behind error-correction codes in
memory circuits and the hierarchical protection and recovery layers which govern filesystems. It is thus desirable to find which, if any, such common denominators that generalize well to comparably more complex systems and scenarios, and, as an intermediate step, which of these mechanisms’ respective properties and benefits than can feasibly be achieved.

**RQ2– To which extent can the drawbacks or caveats associated with virtualization be controlled?**

The indirection implied by virtualizing one or several of the possible resources of a computer (i.e. computation, storage and communication) imposes a certain overhead, one that can be described as the product of the frequency of invocation with the cost of each invocation, and it is therefore desirable to keep these two factors at a minimum as a goal for optimization. For instance, the frequency of invocation can be regulated with the design of the instructions that are eligible by code, making them more semantically dense\(^1\), while the cost of invocation can be reduced by mapping said code to more lower level code sequences that take better advantage of underlying or surrounding components. However, these actions can contradict each other and can impact other qualities in execution adversely, e.g. how easy execution can be instrumented and understood due to the increase in the complex dynamic interactions that are possible. It is therefore relevant to examine how far the adverse consequences reach and the fine-grained causes to those consequences, in order to construct and evaluate tools and techniques for reducing or even eliminating these – effectively improving the value of the virtualization.

**RQ3– Among the sets of tools that enable dynamic instrumentation of software-intensive systems, which are suitable for controlling and monitoring virtualizations in critical software-intensive infrastructures?**

The development, maintenance and usage of software require a hefty amount of tools of varied complexity, irrespective of which level of abstraction the developer, maintainer or user perceives him- or herself to be operating at. A short enumeration of major such tools for development would include the editor, the compiler or interpreter, the debugger, the testing suite, reference manuals, the make system, revision control and so on. Some of these tools are obviously more influential than others, but the moral is that the tasks of the respective stakeholder are very tool-bound and the extent to which these tools can be configured and how they effect the final product or service tends to be misrepresented at best and ignored at worst.

As these tools emerged and co-evolved tightly coupled to the perspective of developing software rather than some larger scope, it is of interest to determine which tools are available and to how large an extent these tools can be used in a transition from software to software-intensive systems to critical software-intensive infrastructures, i.e. how they support monitoring, measuring, altering,

\(^1\)Which can be seen at a low level on CPUs that follow a CISC- design philosophy when compared with a RISC- based approach, but principally similar on more abstract instruction sets.
fixing and tuning complex, live, sensitive and deployed subjects. Additionally, it is also necessary to determine if there are conflicts or incompatibilities between the mechanisms behind the tool(s) and the requirements of the system, and how such conflicts should be resolved or circumvented.

**RQ4– What core services and components are needed to construct experiment environments capable of experimenting with the resilience of software-intensive critical infrastructures?**

Computing has long had the oddity that the arguably best experiment environment for conducting experiments is unsurprisingly enough, provided the computers themselves. As the coordination and control required to raise the ante on what can be examined this way is somehow connected to the development and refinement of managerial facilities such as operating systems, there is well-founded reason for increased concern as to how these facilities influence the final data and the behaviors of the subject. It is nowadays far from a safe assumption that the execution of a single program is isolated and protected from outside disturbance or that its execution will be independent from previous ones made, and the mere act of isolating the factors behind suspected disturbances is far from a trivial matter. Furthermore, when the problem domain is more complex, which is already the case with software-intensive systems of a fairly primitive sort, like web applications and web services, there are already enough variables present to warrant extreme caution. If we then expand the domain further, to also include a legacy-rich secondary structure (such as monitoring systems for the power-grid) there are suddenly several additional complexities brought on by more domain-specific technologies (e.g. programmable logic controllers) combined with a socio-political dimension (liability, negotiated set of interventions that are allowed etc.). At such a stage, the composition and services of the experimental environment itself becomes a legitimate subject of study.

**3.3.2 RQ5– Which scenarios would be useful for both experimentation, training and for generating input to monitoring tools and tuning protective measures?**

The scale and complexity of experiment environment developed for critical infrastructure resilience research combined with the limited resources available means that experiment runs needs to cover as wide a range of interests- and stakeholder concerns- as possible. With the lessons learned from experimenting with infrastructures and from teaching systemic debugging as input, it seems worthwhile to further explore how experiment scenarios should be designed in respect to the active components used, the involvement of human operators, and interactions between the two. With added dynamic and adaptive behaviors in future resilient infrastructures and subsequent extensions to preexisting monitoring systems required to safeguard their operation, operators will be exposed to new and otherwise foreign tools and the infrastructure itself will be exposed to an increased attack surface and possibly, additional attack vectors. Therefore, a combination of training and evaluation scenarios that would ease the gradual introduction of more advanced monitoring tools and at the same time increase
operator awareness of early signs of attack from some adversary, would provide a much needed step towards detailing the successful transition from the brittle infrastructure support systems of today, to the resilient ones of tomorrow.

3.4 Contributions

This section introduces the contributions that stem from exploring the research questions. Fig. 3.1 illustrates how the individual contributions fit together (with the main exception being deliverables to the two European research projects INTEGRAL and SEESGEN-ICT). To further explain this figure, the contributions flow from top to bottom in relation to their approximate kind of contribution ranging from academic/method to industrial/technical and from left to right in terms of time, i.e. least recent to most recent. The dashed line and dotted lines illustrate the connection to the licentiate thesis (*Exploring Software Resilience*) and doctoral thesis respectively, with a dashed line representing an implicit inclusion and a dotted line representing an explicit one. The arrows along with the respective label illustrate the principal output from a contribution, and the connection points are approximations of the relative contribution, meaning how proportionally relevant the input contribution was to the output one.

![Diagram of contributions](attachment:figure_3.1.png)

**Figure 3.1:** Main contributions and their interdependencies.

Table 3.1 is a rough index and classification of contributions along with respective references.
The remainder of this chapter will be used to further contextualize the subset of specific contributions from Fig. 3.1 that comprise Part II of the thesis.

### 3.4.1 Chapter 4 – Self-Healing and Resilient Critical Infrastructures

This chapter serves to provide a more detailed perspective on the prospects and premise covered in Sect. 3.1 through a case-study of the initial phases of the INTEGRAL project, Sect. 3.2 and thus, a high-level view of the targeted application domain.

As per the abstract; *This chapter introduces an experimental approach towards engineering resilient critical infrastructures through the application area of future cell-based energy systems as an example of interacting critical infrastructures supporting energy- and customer based services where resilience is an essential systemic property. We are specifically addressing methods and tools supporting experiment based development and maintenance of resilient software intensive systems through means of self-healing, experimentation environment and hardening of monitoring systems.*
The chapter is directly connected to the aforementioned mission statement and related work, and is an extended version of the publication in [1].

3.4.2 Chapter 5 – Use and Misuse of Virtualization

In this chapter we examine the role of virtualization as a computing mechanism for enabling resilience, which entails models, methods and principles for the controlled experimentation with virtualization in a wide variety of forms. These are primarily derived from historical sources combined with technical examples and cases – alongside the dissection of run-time support systems, virtual machines, interpreters and the likes, both from the world of programming languages and the one of software security. To this end, we place heavy emphasis on the possible benefits and caveats involved and how to successfully harness and control these two central aspects. Thus, this chapter has a strong focus on the interplay between computing philosophy, method and technology and is used as input for the design and implementation of the experiment environment and associated tools presented in Chapters 6, 7 and 8.

As per the abstract; Virtualization is a term riddled with ambiguity. Yet, its various forms are present all-through computing history and together they have essentially become a sort of structural glue that fits various computing pieces together into the complex patchwork that is currently referred to as software. In this chapter, we examine the foundation of virtualization to discern the benefits that can be reaped and the caveats that inhibit its use with the end-goal of improving the construction of future systems and the maintenance of current ones.

The chapter is directly connected to (RQ1, RQ2) and to [70] [2], but was explicitly written for the purpose of serving as a basic model for the other contributions in the thesis.

3.4.3 Chapter 6 – Experimenting With Infrastructures

This chapter covers an engineering approach to the creation of a distributed experiment environment that supports both the management of a SCADA-class system coupled to a prototype microgrid intended for future smart-grid related endeavors. The purpose of this environment is primarily to be able to study the intersection between a specific form of an information processing system and a related information system in a sensitive setting. In order to aid future experiments, the focus is on the underlying structure and problems, both technical and political, that may arise from the development of this, and similar, environments.

As per the abstract; Laboratory environments for experimenting on infrastructures are often challenging both technically, politically and economically. The situation is further complicated when the interaction between infrastructures is in focus rather than the behaviors of a single one. Since ICT often has a key role in experiment management, data gathering and experiment upkeep – controlled experimentation becomes even more difficult when some of the interactions studied are between ICT and another infrastructure. The work described herein concerns design, implementation and lessons learned from the construction of a joint-effort experiment environment for, essentially, experimenting with infrastructures.
3. Structure

The chapter is directly connected to (RQ4) and to [4] [11], and has been included with only minor clarifications, corrections and editing.

3.4.4 Chapter 7 – Retooling and Securing Systemic Debugging

This chapter covers experiences from debugging systemic issues in the context of a debugging task-force at Sony Mobile Communications, gathered partly first-hand from working in such a setting, and through organizing a series of seminars on the subject. It also covers the design and implementation of an open and free prototype tool with two different configurations, one configuration for dynamic and interactive monitoring and visualization tasks and one specialized for debugging, reverse engineering and software security applications. The use of emulators (whole-system virtualization as per Chapter 5) as an evaluation subject for sensor observer-effects is also explored.

As per the abstract; There are a few major principal tools that have long supported the often frustrating and time-consuming part of software development and maintenance that is debugging. These tools are the symbolic debugger, the profiler, the tracer and the crash dump analyzer. With the advancement of dynamic protection mechanisms directed towards hindering or thwarting exploitation of software vulnerabilities (a subset of possible software bugs), combined with a shift from developers being in charge of the development of one distinct piece of software to, instead, piecing a large assortment of third party components and libraries into a common service or platform, many of the mechanisms that the aforementioned tools rely on have been either eliminated, circumvented or otherwise rendered inefficient. In this chapter, we present an industrial case illustrating this shift, highlighting specific issues and challenges facing the effective use of aforementioned tools, then look at how recent developments in tracing frameworks can be further improved to remedy the situation. Lastly, we introduce such a tool alongside initial experimentation and validation.

The chapter is directly connected to (RQ4, RQ3) and to the publications [2,5], and has been included in an extended form with additional figures and more detailed descriptions. The tools in question are also available at [109,110] and the training material used for the seminars can be found in [70], with a brief description in Sect. 3.4.6, below.

3.4.5 Chapter 8 – The Proper Role of Agents in Future Resilient Smart Grids

Concluding Part II of the thesis, this chapter combines the ideas, environments and experiences from the previous ones into a demonstrator, experimental validation and scenario design based on notable recent security incidents from three distinct categories, i.e. Sabotage, Business Intelligence and Business Sabotage.

As per the abstract; One of the major challenges in the transition towards future Smartgrids is a sustainable energy supply through the use of smart energy networks. The implementation of distributed intelligence, supporting governance and maintenance activities, is crucial. This means enabling rapid detection and response to disturbances. To that end, the use of agent technologies for implementing smart self-healing microgrids has been suggested. There are, however, a number of serious cyber security related threats
that also needs to be addressed in order for such a solution to be viable. In this chapter, we identify and describe three such principal threats, alongside a configurable testbed in which harnessed solutions can be assessed and evaluated. We also report experimental results from the use of agents as decision support systems empowering operators. Furthermore we propose a bridging mechanism enabling tailored data-generation and data assessment in not fully specified complex systems such as future Smart Grids.

The chapter is directly connected to (RQ5) and to the pending publication [10], which has been included with slight clarifications, corrections and editing alongside additional figures and more detailed descriptions.

3.4.6 Supplementary Material

Some of the material included is also present in the form of a textbook on systemic debugging, developed in cooperation with Sony Mobile Communications, and while parts have been included in Chapter 2, the majority of work relevant to the thesis have been omitted for the sake of terseness. It is, however, available online through a permissible license (CC-BY-30-Ported) at http://www.systemicsoftwaredebugging.com.

As per the abstract;

Systemic Software Debugging is a light-hearted introduction to the wonderful world of, well, systemic debugging, i.e. chasing down those pesky problems that you will not or cannot find by single-stepping through ten lines of code or taking a peak at the back-trace of a core dump. The kinds of issues that seem to magically appear after software gets even a flicker of a chance to grow out of proportion, when the build system takes a life of its own or when you have to keep a ten year old release of a compiler around just because that was the one version that managed to produce a binary that almost lived up to expectations.
Part II

Contributions
4 Self-Healing & Resilient Critical Information Infrastructures

The layout of this chapter is as follows:

Introduction, Setting the Scene, introduce a case study from the EU project INTEGRAL [105]. INTEGRAL is an acronym for Integrated ICT-platform based distributed control in electricity grids with a large share of distributed energy resources and renewable energy sources. In fact the INTEGRAL ICT platform integrates three different critical infrastructures; i.e., the electric grid including renewables, a customer-oriented business process infrastructure, and a SCADA – ICT infrastructure. The objective of INTEGRAL is to ensure trustworthy behavior during normal, critical, and emergency operating conditions and Understanding and Controlling Complex Systems identify some challenges to that end. Then, Experimental Environments, covers our goals and architecture of experiment based resilience engineering, introducing principles of structured debugging and self-healing. Other approaches provide a brief overview of related international R&D efforts. Lastly, Conclusions assess our approach and findings with respect to some of the challenges given in the Introduction section.

4.1 Introduction

Critical Infrastructures are becoming increasingly complex due to increasing reliance on networking and development of new applications and services. State of the art analysis and road-maps identifying research agendas as well as potential applications can be found in connection with the calls of the EU 7th Research Program (FP7)\(^1\) on Internet of Services, specifically Software and Virtualization. The Objective of future calls has two main research lines. The first research line is Service platforms for the Future Internet and the second one is Highly Innovative Service/Software engineering.

The second research area aims at new technologies and new research lines that will make future innovative service markets highly reliable and trustworthy as well as dynamic. In this context, dynamic means that decisions which, in the traditional software life cycle, were made at design-time will in the future be made at run-time. This means that common software development practices of verification and validation need to be rethought, or at the very least extended, to cover the systemic consequences from run-time reconfigurations.

Current software engineering is based on four key assumptions:

- Dependable large systems can only be attained through rigorous application of the engineering design process (requirements, specifications, prototypes, testing, acceptance).

- The key design objective is an architecture that meets specifications derived from knowledgeable and collectible requirements.

- Individuals of sufficient talent and experience can achieve an intellectual grasp of the system at hand.

- The implementation can be completed before assumed preconditions and requirements changes notably.

The first assumption is challenged by the failures of large systems that used the traditional design process and the successes of systems that simply evolved, e.g., variants of Evolutionary system Development, as the Internet, World Wide Web, and Linux environment. The remaining assumptions above are challenged by the increasingly dynamic environments of our systems, often called information ecosystems; in which large system to a higher degree operate. Furthermore, there is no complete set of requirements because no one individual, or even small group, can have complete knowledge of the whole system or can fully anticipate how the community’s requirements will evolve [12].

Issues related to reliability and trustworthiness of future dynamic service based applications pose many open challenges and Resilience engineering in general, and development of self-healing mechanisms in particular are aiming at resolving some of those challenges (Sect. 4.4).

Most of the systems (work environments) and tasks we are addressing above are thus underspecified at design time. This fact is due to several circumstances such as interactions with a partially unknown environment and increasing dependency on software components (software intensive critical infrastructures). Non-linear phenomena due to unexpected combinations (resonance) of interactions might emerge. Work supported by the systems at hand can therefore not simply follow fixed prescriptions / procedures. Individuals and (virtual) organizations must adjust to the current condition in everything they do (Sect. 4.3.1).

- Success is due to the ability of organizations, groups and individuals correctly to make these adjustments; in particular correctly to anticipate risks before failures and harm occur.

- Failure can be explained as the absence of that ability – either temporary or permanently.

The aim of Resilience Engineering is to strengthen that ability, rather the just to avoid or eliminate failures [72]. An engineering method to strengthen resilience is to identify and implement self-healing mechanisms (Sect. 4.4).
4.2. Setting the Scene

In this chapter we introduce an experimental based methodology towards resilience engineering of Critical Information Infrastructures (Sect. 4.4). Philosophical and conceptual basis are given by models and methods from [76], causation and manipulation models of explanations, [77], and reasoning [78].

4.2 Setting the Scene

Our current application domain, Future Cell-based energy systems, is a prime example of three interacting critical infrastructures where resilience is an essential systemic property to obtain and maintain during operations. The three interacting critical infrastructures are: a Business infrastructure of customer- and energy-based services, a Resilient technical Grid infrastructure, and the Integrated ICT-platform based Distributed Control (IIDC) supporting information distribution and processing. In order to ensure and improve resilience we aim at designing and implementing self-healing mechanisms. Those mechanisms are in turn examples of improving resilience by hardening mechanisms (Sect. 4.4).

The Integral project, summarized in Fig.4.1, focuses on showing seamless integration of DG-RES in physical power systems and commercial power markets enabled by the IIDC ICT system. The focus of the INTEGRAL project can be summarized as providing a configurable information system supporting Active Distribution Grids for integrating Distributed Energy Resources (DER) and is covered by the following success factors:

- Aggregation - Dynamic real-time context, Cells, micro-grids and Virtual power plants.
- Integration - Local distribution grid operations, Higher-level grid operations, markets for energy trading.
- Availability - Practical aggregation mechanisms, Low-cost and industry-quality standard solutions.

The Active Distribution Grids are supposed to support the following Operational Stages:
4. Self-Healing & Resilient Critical Information Infrastructures

- Normal operations - Trading optimization (supplier – consumer, prosumer\(^2\), matching) and Grid optimization (Distribution System Operator - Distributed System/Service Operator (DSO))


- Emergency operations - Self-healing reaction to local faults, Micro-grid restructuring.

The following requirements on ICT Systems for DER Clustering and Aggregation have been identified. The requirements address challenges in Multi-Agent Systems, Distributed Control & Intelligence as well as Electronic markets.

- Scalability - Large numbers of DER components, Large geographical distribution, Complexity limits from centralized control.

- Openness - allowing both current and future DER, as well as innovative PES (Power Equipment Systems) components to connect and disconnect without central control.

- Multi-Actor Interaction - Local and global balancing, Coordination across ownership boundaries, local autonomy.

- Alignment with Liberalized Energy Markets - Different types of regional markets, new market designs.

The Integrated ICT-platform based Distributed Control have the following features:

- Industry Quality; this pertains to the Power System, ICT and business application system dimension.

- Commonly available ICT components and standards; preferably available off-the-shelf.

- Service-Oriented Architectures (SOA) supported by agent technology providing the services.

The introduction of renewable embedded generation, in a dispersed setting, at a large scale and spread over a large area challenges the limits of central control. Central coordination concepts lead to an increase in complexity, system management and cost. Dispersed generation with distributed ICT and bottom-up coordination mechanisms isolates responsibilities and allows decision-making and coordination based on the local primary process connected to the suppliers of electricity. It also allows DER units to connect and disconnect at will and preempts for all (future) DER types. Also, multi-actor interaction requires local and global balancing of stakes and local and global coordination exceeding ownership.

\(^2\)A portmanteau of producer and consumer.
4.3. Understanding and controlling complex systems

boundaries, facilitating decision making locally on local issues and alignment to liberalized energy markets.

In the market design of traditional electricity grids, end-users having shiftable energy or capacity are treated in a similar way as end-users demanding energy at peak prices. All end-user usage is averaged in profiles, according to which costs are attributed following the mix in the development of commodity prices, bilateral contracts and so on in the markets forming the portfolio. Therefore, the full potential of flexibility of demand is not unveiled. Indeed, the way small customers are accounted for in current markets even acts against utilizing flexibility. A similar story can be told for integration of variable output DG-RES resources, which sometimes lead to more carbon emission because of the required extra generation capacity needed to compensate for intermittent fluctuations.

The energy price picture fails to map system costs to the real world, and the same can be said for the distribution costs. As an example, consider having a large HVAC (High Voltage Alternating Current)-related domestic load at peak commodity price periods in moderate climate zones. Effects on the system are accounted for by profiling households and through fixed capacity tariff limits. Fixed capacity tariffs are no problem if they are time-dependent. In novel ICT enabled grids, there are a number of application types on a number of timescales.

Numerous approaches to critical infrastructure modeling and simulation have been explored. A comprehensive survey conducted in 2006 of current solutions highlights several of these approaches [13]. A more recent approach to critical infrastructure modeling and simulation focuses on the development of a coupled modeling approach as described in [14, 15]. Under this approach, individual infrastructure models are integrated in a generalized way with models of infrastructure dependencies to enable system of system analysis – thus coupling the fidelity of individual infrastructure models with the requirement for situated analysis. Fig. 4.2 gives our four-tiered architecture for INTEGRAL experimental environments.

The promise of a coupled approach to critical infrastructure modeling and simulation highlights the challenge of designing a proper integration framework. Specifications of such frameworks have been developed. For example, the IEEE Standard 1516 High-Level Architecture (HLA) [91] for modeling and simulation represent such specification. The HLA specification is comprised of federates (which could model individual infrastructures), an object model (which defines a vocabulary for discourse among federates), and a run-time interface and infrastructure (which enable interaction among federates). In INTEGRAL each of the three Field tests could be regarded as different configuration of the three critical infrastructures involved.

4.3 Understanding and controlling complex systems

Critical infrastructure information systems are complex and hard to analyze for malfunction and other unwanted behavior. Assured resilience of such systems is thus a challenge. This difficulty stem from several properties, both from the actual information system and from the physical infrastructure. In this section
4. Self-Healing & Resilient Critical Information Infrastructures

![Figure 4.2: A four tiered architecture of INTEGRAL experimental environments.](image)

we introduce some general aspects of complex systems in Sect. 4.3.1. In Sect. 4.3.2 we specify our focus of investigations to software intensive systems.

### 4.3.1 Complex systems

There are several models and theories related to complex systems. For our purposes we focus on the following two important aspects related to understanding and controlling systemic behavior. The concepts are coupling and interactiveness between system components.

The coupling between components can be in the range of loose – tight whence the interactiveness could be in the range of linear – complex. On characterization of complex systems is that the interactions involve feedback loops and tight couplings. Typically, we have complex systems when we do not know how to produce the output through linear systems [72]. The complexity of complex systems is often due to some intractable system criterion such as:

- Principles of functioning are unknown or only partially known.
- Description of systems is difficult and contains many details.
- Description takes a long time to make.
- Systems change before the descriptions are completed.

To address risk analysis, accident models, and control of (complex) systems linear and non-linear models have been developed. Linear models were typically developed before 1984 [79]. Whence non-linear models, such as the Functional Resonance Analysis Method – FRAM, are more recent [72].
4.3. Understanding and controlling complex systems

Linear models of accidents and risks assume that accidents are the (natural) culmination of a series of events or circumstances, which occur in a specific and recognizable order. As a consequence, finding and eliminating possible causes prevent accidents. This can be achieved, for instance, by strengthening barriers and defenses between components (Sect. 4.4 and Sect. 4.6). Furthermore, improving the ability of an organization to respond ensures safety and security.

To address challenges of future network based systems such as those described in Sect. 4.1 and our case study INTEGRAL systems in Sect. 4.2 we have to also model and control behavior in non-linear systems.

Non-linear phenomena have to be assessed when large scale systems stretch established methods to the limit and as human and social factors are recognized as important contributors – both to accidents and to safety. As a matter of fact, non-linear accident models, such as FRAM, address some of the challenges identified in Sect. 4.1, related to assessments of underspecified systems and hence to the challenges of Resilience Engineering.

Our approach of harnessing non-linear phenomena and intractability is to identify local linear component interactions by context dependent recursive boundaries in our experimental investigations Sect. 4.4 [80].

4.3.2 Software-Intensive Systems

Software is a key component in network-based systems. Development of methods and tools supporting building and maintaining software intensive systems is thus in focus of current R&D efforts worldwide.

Current means and methods for developing software in the sense of isolated systems that generate and process information are largely able to cope with most of the characteristics of software; characteristics such as feedback loops, a large space of possible states, recursions, etc. This is made possible through precise, well-defined sets of inputs (interaction-patterns) and expected outputs (results) i.e. requirements, but also by verifying and validating each step of the development process. With this approach, it is of utmost importance that the requirements put forth corresponds to the actual interaction-patterns in the final system, otherwise the requirements are rather assumptions on behavior, and the results of any deviating interaction-patterns are undefined. Consequently, software enforces requirements.

Source-code, design documents and requirement specifications are descriptions of intended software systems. Software itself is merely a series of instructions, which are capable of being executed on a machine. These descriptions are transformed into software through an array of finely tuned tools, which, to a large extent, makes the output coupled to the targeted machine. This machine might be virtual (composed out of other software), hardware or a combination of both. The combination of supporting software and enabling machinery constitutes the software’s environment. This environment, amongst other things, provides the software with input, but also modifies and optimizes the software in order to take better advantage of current conditions – ultimately making software dynamic and adaptive.
Furthermore, software and its environment combined still do not function as an isolated system, but are largely affected by those who, directly or indirectly, operate them. Due in part to the necessarily tight coupling between software and its environment, software is inherently vulnerable to cascading effects and therefore brittle. Subsequently, changes to any of the components that make out the environment may potentially break assumptions software have on environment configuration, rendering the software broken – thus we need to have software that adapts to requirements.

Lastly, the current dependency of software as parts in a larger information-processing systems within an infrastructure, effectively expand the software’s environment to cover not only the local machine and supporting apparatus, but other components partially outside the scope of our control. Hence, for requirements to be enforceable, they too need to be adaptable and reflect the current systemic condition - because having a dynamic system (software) adjust in accordance with requirements that cannot reasonably be fulfilled (since they no longer reflect the current state of the environment) will ultimately fail.

Therefore, it is reasonable, in the context of critical infrastructures and similar distributed systems, to model software not as isolated software-systems but rather as software-intensive systems wherein software acts as structural glue to information processing in a heterogeneous complex larger system, something that introduces new challenges to software development and maintenance:

- Single-instance – An often-exploited property of software as isolated systems is that their operating conditions can easily be reproduced and have execution repeated (under certain conditions, even reversed or reverted) with little to no effort and comparable outcome. This allows us to ascertain system parameters (state, performance, etc.). The software-intensive systems of critical infrastructures are not only single-instanced, unique, but cannot reliably be taken down or re-instantiated, partly because stakeholders depend on the existing instance, but also because of the circular inter-dependencies between infrastructures which effectively preventing a full-scale black-start.

- Incoherent world-view – Obtaining an operational view of overall system state becomes increasingly important when there are no other ways of ascertaining operating conditions and system behavior. However, due to the heterogeneous nature of software-intensive systems different parts and component process and communicate at different, possibly unknown, rates making them hard to sample in a comparable fashion.

Because of these challenges brought on by software-intensive systems it is of utmost importance that problems in one particular piece of software do not propagate and cause resonance within the system.
4.4 Experiment Environments

In this section we address two aspects of design and maintenance of resilient software intensive systems. In Sect. 4.4.1 we outline some fundamental aspects of experiment environments to enhance resilience. In Sect. 4.4 we address self-healing and resilience. A key purpose is to develop and test self-healing mechanisms and enable proper monitoring – in order to ensure resilience.

4.4.1 Resilience Experimentation

The key feature of our experimental environment is that we can improve resilience, even if we have globally intractability or non-linear phenomena, by supporting local linearization (recursive boundaries) and selected virtualizations. In short, models, methods and tools supporting structured and context dependant systemic debugging.

With the challenges put forth by software-intensive systems, we consequently face the problem of instrumenting existing executing systems, but also to take advantage of lessons learned in regards to software problems and alter these systems accordingly. This involves isolating and modifying problematic regions between necessarily occurring borders, i.e. virtualizations. This also means that there is an immanent need to be able to adjust environment properties to evaluate the consequences from systemic reconfigurations and hopefully expose.

As there is considerable risk of adverse effects with this kind of procedure, it is not reasonable to expect to be able to directly work with the subject software-intensive system without first ensuring that the intended instrumentation is reliable to as large an extent as possible.

For these two reasons, there is a direct need to experiment not only with software but also its environment to the extent that we can reliably obtain feedback on the behaviors and properties of the subject that is both accurate and generalisable to a large-enough degree to be representative of the targeted system.

Towards these ends, our experiment environment consists of two principal modes of operation both supported by a set of tools that allow for gathering of measurements through a reconfigurable network of dynamically created probes injectable into interfaces at borders. The modes are thus:

- Isolated - which models the subject operating in critical or islanded mode where links to other parts of the system have been severed.

- Distributed – where two separate environments are connected through one or several VPN- protected links running on open public communication channels, corresponding to the case provided by INTEGRAL.

4.4.2 Self-healing and Resilience

The well-known N-1 criteria in power system operations can be seen as a method of self-healing that could be implemented by different mechanisms. The N-1 criteria in electric transmission or distribution systems ensure that loss of one link
does not cause failure elsewhere in the system or loss of service. In INTEGRAL Field tests we introduce the concept of agent enabled self-healing of the virtual utility (Section 2). In fact, we are specifically addressing self-healing in software intensive controlled complex equipments and systems.

Self-healing, as a concept, has a long history in computing. Previous efforts have mainly been related to adaptation mechanisms in operating systems or multiprocessor systems. Recent interest in self-healing however is due to the increased complexity of our software intensive systems [16, 17].

Modern practical computing systems are much more complex than the simple programs on which we developed our models of dependability. These dependability models typically rely on precise specifications, which is in practice impossible to obtain for larger distributed systems.

Self-healing could be defined as a mean to transform brittle tightly coupled systems into loosely coupled ductile systems with flexible interaction patterns (virtualization). The idea is that the flexibility of interaction could absorb (self-heal) disturbances not foreseeable at design time of the system. Having said that, it is paramount that self-healing mechanisms are engineered based on input from carefully performed experiments.

The degree of self-healing of a systemic fault or failure could be measured in terms of automation. At one end of a spectrum self-healing could be monitoring by a system operator given the appropriate support tools. At the other end of the spectrum self-healing tasks are fully automated. We have thus either human supported intelligent systems or automatic intelligent systems. In the INTEGRAL field tests we will have both kinds of self-healing support.

### 4.5 Other approaches

Our experimental environment (Sect. 4.4, Chapters 6 and 8) has similar approaches and goals as the NSF GENI [111] efforts (Fig. 4.3). Arguably, modeling, understanding and maintaining correct information flows is fundamental for ensuring proper behavior of critical infrastructures [81]. From Figure 4.3 we can read that there are indeed different types of information, i.e., measurements, control information and user information involved in the systems we are addressing.

![Figure 4.3: Architecture of the NSF GENI experiment platform.](image-url)
The EU STREP project INTEGRAL is a follow up of the earlier EU projects CRISP and MicroGrids. According to the Strategic Research Agenda (SRA) of SmartGrids [112], standardization, modularization and programmable functionality will enable an economy of scale of future power systems, potentially leading to lower costs of operations and more expandable systems.

4.6 Conclusions

We have outlined and motivated two tools and environments supporting a structured experiment based approach towards hardening critical infrastructures [18–27]. The case study is the EU project INTEGRAL focusing on resilient integration of DER/RES in virtual utilities. The mitigating of vulnerabilities is supported by an engineering approach utilizing a combination of virtualization and self-healing techniques.

Our efforts on self-healing mechanisms have also been on the low and high levels of system interaction. That is on securing software execution by hardening mechanisms and self-healing at the mission level [22, 23]. Some preliminary work on self-healing and resilience is reported in [1]. The purpose of the tools and environments introduced in Sect. 4.4 is to further identify and implement self-healing mechanisms, in a principled way, at remaining system levels.
5 Use and Misuse of Virtualization

The layout of this chapter is as follows:

*Setting the Scene* covers trends and changes in the components and scope of software intensive systems. *Approaching Virtualization* then depicts applied virtualization, but as a primary mechanism found in live, executing, software, rather than, e.g. a means for running several guest operating-systems on a single computer. In *Possibilities*, a wide assortment of virtualization benefits are briefly covered. In *Caveats* it is argued that there are inherent risks and complexities that follow with the possibilities discussed and which ultimately need to be accounted for. Lastly, in *Moving Forward*, several approaches to account for some of these risks are suggested.

5.1 Setting the Scene

That computers have become key components in the controlled processing of large quantities of information is a fact. Considering the short timeframe during which digital computers have been available, this development is not only impressive but also a testament to their versatility and potential. The technical development is particularly interesting, not because of the increase in clock frequency from hertz to gigahertz or in storage capacity from kilobytes to terabytes, but rather because of the transition from one-shot automated calculation to dynamic and adaptive heterogeneous systems where key information can only be extracted during run-time.

In a similar fashion, the main task of a programmer has also shifted somewhat, from being focused on the implementation of a few selected data structures and algorithms to stitching together components of varying levels of abstraction, provided by large frameworks and libraries of functions, into a coherent, solution targeted to some specific need or purpose. This latter challenge, we reckon, is by far the most complicated one; a challenge illustrated in part by the apparent need not only for sophisticated programming languages for describing an intended system but also by the collection of tools needed to piece together and produce the software in its final static form.

Furthermore, the hardware involved cannot be considered as merely a compact version of the ancient behemoths. Computers have not merely become more compact while growing in capacity, they have also been been complemented by, amongst other things, large assortments of specialized processors designed to explicitly off-load heavyweight calculation, enforce various forms of protection etc.
Most of these auxiliary processors, generic or specialized, can be programmed to a certain degree and strong benefits can be gained if these are coordinated optimally. With firmware, microcode and other low-level instruction formats that are at least partially modifiable and also allow components with a previously fixed behavior to be adaptive and dynamic, the age-old distinction between hardware and software no longer seems that relevant. Thus, the refinement and advancement of efficient programs and services that rely on such separation are probably not the most productive ways to move forward. Even the embedded systems that for the most part could initially be considered isolated and separated, with clearly defined roles and responsibilities, exert dynamic qualities, which is illustrated in the rapid development in cell-phone platforms and technology [2].

For these reasons, the dynamic side of software execution has become the focal point of a lot of interest. However, the dynamic software landscape is complicated, and made possible only through many layers of advanced support where dynamic linking, garbage collection, reflection and similar technologies can be considered almost anatomical\(^1\), features. In addition, source code, being the most widely used causal model for understanding the detailed behaviors of a specific software system, explicitly hides these features and can only be considered a primitive model for the executing software, at best [28].

Take the crude description of the dynamic side to computing above, and add to this the wide-spread communication technologies that characterize the internet and the world wide web. These technologies further push the envelope by stripping away locality so that programs or smaller pieces of code can be pieced together from essentially all over the globe, or allowing a computing task to be split and divided across several computers in a seemingly transparent fashion. However, the problems and dependencies that come with these technologies do add up, making it increasingly difficult to predict future system behavior or even simply obtaining an accurate overview of the parts and pieces involved. This forces us to treat some software-intensive systems as a combination of large, complex, open, dynamic, heterogeneous and concurrent processes.

All the above-mentioned factors and shifts combined serve as a major incentive for improving the ways in which we analyze, maintain, improve and experiment with systems of this nature. This chapter entails an important enabling mechanism for such tasks, virtualization, a mechanism which we wholly depend on at a very fundamental level.

### 5.2 Approaching Virtualization

This section starts with a rough definition of virtualization. The aim is to untangle it to the extent where it becomes possible to discuss benefits, risks and productive ways of taking advantage of respective capabilities, not in the sense of developing software as such, but rather to be able to understand (reverse-engineering) and refine (optimizing, adding features and correcting undesired behaviors) the sort of complex software-intensive systems that were depicted in the previous section.

\(^1\)Anatomical in the sense that no matter what software-subject is being dissected and studied, some specific components are very likely to be found.
It should be noted that the discussion which follows is broader and more generic than some more specialized cases that are also referred to as virtualizations, running several operating systems (guests) inside another (host), Fig. 5.1. This particular form of application will not be covered in detail here, but such discussion can be readily found elsewhere [29, 30].

The basic definition of a virtualization is *the abstraction of computing and its resources*. Judging by the range of publications on the subject, there seem to be at least thirty or so commonly added prefixes such as para-, network-, platform-, resource- and so on, all used to further highlight or emphasize some particular aspect or property. While it seems quite possible to establish a taxonomy of these different types and subtypes, and the overlaps and ambiguities involved seem to warrant the research and development of a comprehensive one, but this is far beyond our intended scope. It is likely, however, that the definition and perspective that this section stipulates apply to a larger assortment of work on virtualization, although making such a contribution was not our goal.

Deconstructing this definition, computing is here modeled as:

\[
\text{computing} = \text{code} + \text{execution} \tag{5.1}
\]

Using the von Neumann architecture, as per (Fig. 5.2), the resources that can be subjected to abstraction become clear and they are thus: *storage, communication* and *computation*.

A virtualization is established by determining which of these components that are to be virtualized. The particular case where all three are being covered, is referred to as a *whole-system virtualization*, better known as a *virtual machine*. On a higher level, it can be said that the act of virtualizing one or more components of computing in effect *de-couples* code from the semantics and syntax of one machine (or parts thereof), and *re-couples* it to another one. It then becomes the responsibility of the virtualization to implement the *translation* between these two formal spaces.
5. USE AND MISUSE OF VIRTUALIZATION

To exemplify this procedure, consider the program, a, which has been either written for, or compiled to, an instruction set native to a specific machine. The instruction set definition covers not only the state transitions that each instruction will perform, but also which exact binary sequences that correspond to which instruction, i.e. its representations. If a machine operating using a different instruction set, b, would be configured to execute this foreign code, it would most likely output an illegal instruction error or, in a densely packed instruction space, unproductive state transitions. If we, instead of trying to execute the code intended for a directly, write code using the instructions of b that decode the sequences corresponding to a and maps these to a corresponding version in b, we have essentially decoupled the program from the first processor, added an indirection (which performs the translation) and in effect, formed a (partial) virtualization of the computation performed.

By finding the mechanism(s) through which a subsystem communicates with other parts of a system, we can identify the interface(s) between components. An interface can be seen as the dimensions and boundaries of data exchange. If there are rules which impose restrictions on the flow across the interface and that either side of the interface can verify and act upon (enforce), we have a protocol. However, if there is an established flow which is implicitly assumed (not enforced) we instead get a convention. An example of this distinction can be seen by comparing the implementation of C- style function calls (within a process) to operating-system calls (between process and kernel), where the former assumes that certain data are to be present on the stack in a specific order, but does not specify how it should be placed there, while the latter is likely to require a regulated, stricter set of conditions in order to be invoked. The benefits of distinguishing between the weaker convention and the stronger protocol become clear in a situation where states and flows need to be analyzed, as it is considerably easier to monitor and enforce trigger conditions on a flow with a precisely articulated structure than to try and filter out all possible ways that a certain state transition can be achieved. Merely identifying the presence of a local function call that follows one of the handful of C- calling conventions out there can be a difficult task, as illustrated by [113].

---

2An instruction set where most or all of the possible binary sequences have corresponding instructions.
Taking advantage of the interfaces which connect subsystems together, these are the places where a selected virtualization can be implemented. The requirements are quite simply that it is must be possible to inject code that intercepts all interface points relevant to the protocols and conventions in play. It is, in general, the properties of the injected code that will regulate the benefits that can be achieved. In one sense, the main activity of this indirection\(^3\) is, as previously stated, the dynamic translation between formal spaces. However, the actual extent of this translation does vary with the complexity and size of the protocol, the nature of the virtualized components, and the merits of the underlying layer. To further distinguish between forms of translation that an implementation of a virtualization can perform, *emulated* and *simulated* (or modeled) translations are here considered as different enough to warrant the distinction. In the case of emulation, there should be one or more existing implementations that take precedence in regards to ambiguities, imprecisions and conflicts in the protocol(s) involved. A proper emulation should therefore take such variations into account. By contrast, a simulation can be considered as a limited, single interpretation of the protocol(s), one that only has to resemble parts of the protocol(s). This greatly reduces the number of virtualization benefits that may apply.

In essence, what a virtualization aims to establish, as illustrated by (Fig. 5.3), is the subdivision of a state space, (physical OR virtual) into essentially two parts, a virtual space and its respective (pseudo-) machine. The latter term is used to illustrate that during execution, the virtual space is necessarily dependent on the operation of its machine(s). Events that affect the machine can therefore propagate to the virtual space. Putting this fact aside for a second, the optimal situation for a virtual space is shown in (Fig. 5.4) where the execution-flow through the program is solely controlled by the input available to the machine upon execution. Each executed instruction imposes a transition in the state space from one state to another. Whenever this reaches a point where there are either no more instructions to be executed, or the current state is such that the machine will interpret

\(^3\)There are corner cases that do not fit particularly well into this model, like the code swapping and trampolines that are part of on-demand linking.
5. USE AND MISUSE OF VIRTUALIZATION

Figure 5.4: The virtualization ideal.

It as a reason to terminate execution, and the system will be brought to a post-execution state. A major property to emphasize here is that of repeatability. Given the same input to the same program, the output discernible at the post-execution state will be identical between repeated execution runs of the program.

As suggested by the initial definition, there is a certain overlap between virtualization and the comparably broader notion of abstractions. While abstractions can be considered as being ideas distanced from objects, computing abstractions such as a function, a procedure or a method may well be modeling artifacts that do not have to be present during execution. By contrast, virtualization in the sense described here, concerns the embodiment of an abstraction. As such, its mechanisms can be both measured and altered and do have influence on the state of the larger formal system. Compare, for instance, the case of a function written in C that has been declared extern to one that has been declared static. In the extern case\(^4\), the compiler is forced to emit code to deal with invoking a function while in the static case, the compiler can remove the encapsulation of the function entirely in the name of optimization.

From these definitions of virtualization, interface, protocol, emulation and simulation, there are a few conclusions that can be inferred as to the merits of the specific phenomena which they reference;

- A virtualization by itself, cannot provide additional capabilities that surpass the respective capabilities of the machine that enables its execution.

- It is unlikely that there are zero or just one virtualization involved in the execution of a program on a computer. Rather, we are dealing with possibly large and complex hierarchies of nested virtualizations.

To summarize this section, a virtualization is considered to be the code patterns that implement the translation between a computing abstraction and its underlying machinery, which itself could be a virtualization. This implies an indirection in terms of the access to one or several of the primitive resources of a computer, i.e. communication, calculation and storage. The virtualization imposes a split of the state space of the machine (the native instructions) into a virtual space and

\(^4\)Or perhaps even more distinct, when calling a function through a pointer.
a machine space. A partial evaluation criterion in this regard is that code bound to the virtual space will not execute in the space of the machine without the explicit translation and governance performed by the virtualization. Data and code exchange between the two spaces is bound by the respective interfaces, protocols and conventions used for communicating with the virtualization.

5.3 Possibilities

It is something of an optimization adage that the computing that is neither needed nor performed, is the quickest form of computing. In other words, there is no point to implement things that do not serve any explicit function. As any virtualization implies one or several indirections that do not benefit the executing system as such, it is reasonable to expect some sort of tangible benefit that motivates implementing and maintaining a running virtualization, apart from soft preferences such as aesthetics. This section is used to highlight a rather wide, if not conclusive, assortment of goals that virtualization is instrumental in achieving. A more brief, but similar assessment can be found in [31].

Compatibility is arguably one of the stronger such possibilities of virtualizing computing resources. It is often touted as a hallmark benefit of programming languages specifications that include not only the semantics and syntax of the language, but also the characteristics of the execution environment it is intended to run inside, using selling points akin to acronyms like Write Once, Run Everywhere (WORE). Apart from such design-phase foresight, a more direct form of compatibility can be added to already deployed systems in the form of whole-system virtualization implemented using emulation. This has been most powerfully used on software built for machines that have, for some reason, fallen out of favor (backwards compatibility), but is used also when writing code for future systems where the end hardware is unavailable or accessible only in limited quantities. The difficulty of the task in implementing such a virtualization depends greatly on the complexity of the protocol(s) involved and can be found at different levels of abstraction when looking at larger projects such as Multiple Arcade Machine Emulator (MAME) [114] alongside the Wine Is Not an Emulator (WINE) [115] Win32 API re-implementation, illustrating both the difficulties and the varying overhead involved.

Adaptive optimization is an interesting opportunity and the subject of much work in the direction of specializing a computing task to current runtime conditions [32]. It can be applied to the virtualization itself or to the virtualized space. In a traditional sense, optimization has been a major focus for static analysis techniques, particularly as an important step in static translators such as compilers. As previously static decisions are being pushed into the run-time realm, decisions on where and when to apply various optimization techniques will also have to be done at run-time. Then, you get access to more advanced and target-specific

\[5\]While interesting in some regards, such factors are not considered in this thesis.

\[6\]Also referred to in some settings as mobility or portability even though these terms are not entirely synonymous.
systemic optimization strategies\textsuperscript{7} that use operating system level information on current conditions as part of the heuristic. One of the more versatile efforts in this direction is the Low-level Virtual Machine (LLVM) project [33].

Resilience is the ability of a system to reconfigure itself to minimize or harness disturbances and can be seen as both a complement and a contrast to fault tolerance and dependability. An example to this effect can be found in the comparison between two software systems which both implement some kind of multitasking between programs where one of the systems use virtualization in the form of process separation while the other system instead makes sure that the programs are cooperatively multitasking, in the sense of coroutines or similar constructs, so that they are able to yield execution and other resources back and forth in a cooperative manner. In such a system, we run the risk of some kind of error (say a live-lock preventing execution to be yielded or a wild memory write hitting an invalid address or data belonging to another task) that adversely affects not only the system-specific task but cascades to other tasks as well. For the system, however, the live-locked process can be preempted so that the memory writes will either hit pages belonging to the process or unallocated pages, generating a trap which can be acted upon. Ideally, the other processes should go on unscathed. This illustrates a horizontal form of protection, meaning that parallel instances of the same virtualization are kept separated. Consequently, vertical protection guards against some unwanted problems that may exist in lower hierarchies of the same involved resources. To exemplify this, consider RAID [8]. In RAID-based storage philosophy, a common abstraction such as a file or a device gets a configurable translation (RAID level) which distributes read and write requests across several devices to form a virtual volume. This can be used with a parity component for the stored data to survive a certain amount of device failures, or to improve performance.

Software Security is to a large extent centered around the idea of being able to separate, prioritize and regulate access to the resources of a machine, and is in this regard an interesting dimension of virtualization-based separation. That which characterizes software security is, however, not this particular demarcation in itself, but rather the presence and the likely consequences of a conflict of interest between the stakeholders of a system and an assumed antagonist, where the many flaws in the interplay of a computer and the code it runs become the playing field where this conflict is acted out. The antagonist(s) actively search for the specific flaws that can be used (be exploited) to take control of the machine in its entirety or of some interesting subset. Meanwhile, the stakeholders work towards finding and eradicating these flaws. A direct consequence of this arms race is that many of the actual problems that hide behind the implementation of a certain separation is brought out into the open where it may serve as a formidable basis for the study of virtualization mechanics.

Maintenance is a considerable part of the later stages of a software’s life-cycle. Normal maintenance tasks, e.g. system administration, cover things such as installation, applying security fixes and other forms of upgrades, but also debug-

\textsuperscript{7}This is more fine-grained than, for instance, the compilation directives of optimizing either for speed or for size.
ging. These are probably the more prevalent and prolific uses of virtualization technologies. One major such technology revolves around snapshotting, i.e. storing the virtual space in a form that can be analyzed offline, distributed and re-instantiated on other machines, or reverted to in case of failure. Attempts along these lines are often creative, like skewing time on emulated embedded systems so that they can execute on a faster machine than the target platform, essentially aging the virtual space more quickly than the targeted physical hardware would allow as a means for getting better test-cases or lower the mean time to a specific failure.

In the end, the benefits one might gain from virtualizing parts or the whole of the targeted or intended system will, of course, vary somewhat depending on the specifics of the system and the application domain. However, while it is likely that some virtualization techniques can be embedded into a system fairly easily, it takes considerably more preparation to make use of all of them, even though this may be possible.

5.4 Caveats

A well-known quote attributed to David Wheeler is that "All problems in computer science can be solved by another level of indirection" with the equally arguable response being something like "except for the problem of having too many levels of indirection". This is a statement which captures the misuse of virtualizations fairly accurately – and it turns out that the problem of having too many levels of indirection may well surpass the problem these indirections were meant to resolve.

This section primarily illustrates that although virtualization and associated techniques can be added and embedded into any computing system with relative ease, the benefits that they bring may come at a considerable cost. The section also describes the most important problems that need to be addressed. These problems correspond to the sections in Possibilities, but are differently grouped: External and Persistent states, Dynamic processes, Performance, Homogeneity, Software Security.

5.4.1 External and Persistent States

Expanding on the abstract figure of the virtualization ideal (Fig. 5.4), we get (Fig. 5.5) where the two major additions are the notions of persistent states and of external states. This highlights the principal challenge for any virtualization effort: in order to successfully take advantage of the presented possibilities (without adverse side effects), central dependencies in these two categories need to be controlled, even eliminated, if possible. To clarify: persistent states refer to data which somehow influence the execution of a program and that are not part of the static input, configuration. As an example, using process separation as a vantage point, a persistent state would be something akin to a database where a program can store configuration options that can be written to and/or read from intermittently. Thus, data stored will persist after a program has been terminated. In a

8Not all external states are persistent but all persistent states are by definition external.
closely related fashion, we have other external state holders which may be shared between different processes, such as file- and socket- descriptors and pipes. The principal criteria is that a state holder is closely tied to something that resides outside the virtualized space.

For a more programming-centric example, consider a function from an imperative language such as C in relation to the virtualization-ideal model. Its inputs are the arguments that can be passed to it and its outputs are the value(s) that the function can return. Provided that we enter/have the same arguments, the outcome of the repeated executions of the same function, whatever its purpose, should return the same values. However, if the function uses the value of a variable declared in a dynamic scope for input to a calculation, or a conditional expression at some point during the course of its run, these accesses will also need to be controlled in order to ascertain that repeated executions will return the same output. The dynamically scoped variable is thus an external and persistent state holder within the context of a certain function implemented as a virtualization. Consider something more representative of real software than this short example and the number of external and persistent state holders is suddenly vast. Some of these state holders are inherited from the execution environment and the operating system while others emerge from the interaction between programs and between programs and their auxiliary systems.

As a rule of thumb, the code present in the virtual space is either larger than or equal to the corresponding code that will be executed in its respective machine space, giving the instructions a multiplicity of \textit{one to one} or \textit{one to many}. This also relates to the performance model (Eq. 5.2, Sect. 5.4.3), but has other consequences as well, for instance the ability and granularity with which one can instrument and relate measurements gathered to code that is assumed to be causally relevant for said measurements. This problem is related to one of the challenges in constructing debuggers with source-code level symbolic representation. For each statement in the source code, there can be 0..n corresponding instructions in the

![Figure 5.5: The fundamental problem.](image-url)
5.4. Caveats

final code\textsuperscript{9}. In fact, these instructions do not have to be exclusively bound to only the corresponding statement in the source code, they can be reused wherever reasonable, reducing the accuracy and usefulness of the breakpoints.

It has previously been established that one intended effect of virtualizing one or all resources should decouple the instructions that correspond with the use of these resources and recouple the instructions to the protocols of the virtualization. This would, in effect, render the specifics of the resources entirely opaque from the perspective of the virtual space. If this is not the case, a program could be constructed with the instructions of the virtual space that would depend not only the presence of an implementation of the protocols in play, but also on the specifics of the underlying machinery producing a consequence which is counterproductive to several virtualization goals. As covered from a security context in [34] and elsewhere, it is, however, difficult for a virtualization to hide itself in such a way that a program executing inside the virtual space can neither detect the fact that there is a virtualization present nor make out specifics of the underlying machine. This means that there is also the risk of a program being coupled to both the virtualization and a machine.

5.4.2 Static Versus Dynamic Processes

A lot of effort has been put into both development processes and tools in order to affix descriptions (which is source code) of intended program behavior to the behavior of the final code that will be executed. This is an important link to maintain in order for these descriptions to function as predictive models for actual program behavior during execution. Unfortunately, even for languages where the run-time support system can be made very minimalistic, such as the case with C, there is a considerable discrepancy between what the source code describes and what is actually being executed. This is the case even before we consider the full involvement of build systems that coordinate and configure the large array of tools involved.

As the virtualization efforts get more involved when combating the other problems described here, this discrepancy widens, further diluting the relevance of source code as a predictive model for runtime behavior. As an example to this effect, consider (Fig. 5.6) in relation to (Fig. 5.7) in the sense of the events they describe in comparison to what execution of the code will accomplish. Although these figures have been constructed to specifically illustrate the problem, the actual occurrence of the underlying pattern can also be quite readily found in production code, but in a slightly less obvious form. In (Fig. 5.6), we cannot discern much about the execution before considering the activities of the dynamic linker, for which in turn we need data on environment configuration such as the LD\textunderscore PRELOAD environment variable used by, amongst others, the GNU LD linker [116]. In the (Fig. 5.7) case, things go even further. Not only does the (Fig. 5.6) still apply, we also need to know the contents and exact order of what was being passed through the Transmission Control Protocol (TCP) socket. As the word implies, the \textit{dynamic} case requires one to consider a time-frame as well;

\textsuperscript{9}Zero matching instructions when the statement has been removed through subexpression elimination or similar form of optimization.
5. USE AND MISUSE OF VIRTUALIZATION

```c
#include <dlfcn.h>

int main(int argc, char* argv[]){
    static int (*fp)(void);
    int* handle = dlopen("input.so", RTLD_LAZY);

    if (handle){
        fp = (int (*)(void))dlsym(handle, "infun");
        if (fp) fp();
    }

    return 0;
}
```

Figure 5.6: Dynamically loading a shared library (in C), searching for the symbol ‘infun’ and handing execution over to the corresponding code injected by the dynamic linker.

there is not only a where and what but also a when to consider. This is illustrated by the on-demand linking feature of dynamic linkers. On-demand linking means that when a library is loaded, the actual code for all functions is not necessarily loaded into memory. Instead, placeholder code will be loaded at referenced addresses which, when executed, will link in the real code\(^\text{10}\) [82]. These dynamic aspects at all times when there is a virtualization present that, for any reason, needs to be analyzed and instrumented.

```ruby
require 'socket'

TCPServer.new(8080).accept.each_line{|a|
    eval(a.strip)
}
```

Figure 5.7: Scripted dynamic loading (in Ruby) where each line of text sent by a client connected to the running program will be reinterpreted as code in the virtual space.

5.4.3 Performance

Undoubtedly, there is a certain overhead when performing the translation between formal spaces during run-time, especially if the instructions are to be monitored and verified as well as per the distinction between convention and protocol. The question then becomes, if (or when) this overhead is negligible or not.

Because of the sheer amount of adaptive processes which influence performance one way or another, there is a strong incentive to include code that account or compensate for changes in the environment (operating parameters), throttling

\(^{10}\)This process can, however, be used to intercept and hijack execution. A mechanism which surreptitious software and run-time instrumentation techniques often take advantage of.
CPU frequency, as suggested in [83]. This is especially important in embedded and mobile systems where performance problems concern notably finite resources such as remaining battery charges.

As we deal with several levels of interconnected abstractions, and it is difficult to establish accurate and reliable metrics, we go no further than a tentative rule of thumb, e.g. the model in Eq. 5.2.

\[
\text{overhead} = \text{invocation frequency} \times \text{translation cost}
\]  

Note that in this model, we do not consider the actual cost of the computation itself. The performance consequences captured are closely related to the problem described in Sect. 5.4.1 on the density of protocols primarily exposed to the virtual space. Compare, for instance, between the use of direct translation (interpretation) between two different CPU instruction sets when employing virtualization implemented using emulation in order to achieve compatibility and using some optimization technique such as dynamic translation (aka. dynamic recompilation, just-in-time compilation, etc.). This is a well-known and expensive operation as each instruction in the foreign instruction set requires code which decodes said instruction and maps it to the most reasonable instruction(s) in the native instruction set, while at the same time accounting for other differences between host and foreign CPU, including registers and memory model. Even for fairly primitive foreign CPUs, the translation cost alone can be a factor over several hundred times compared to the assumed minimal 1 to 1 mapping. In addition, the invocation frequency of the instructions may even be higher than the actual clock of the native CPU, although this is probably a rare phenomenon. With dynamic translation, the translation cost is gradually lowered as invocations are replaced with the result of earlier translations, either directly or after a set number of invocations have occurred. The downside is that this can have the opposite effect on highly dynamic code. For some applications, there is also a, more uncommon, third option in the form of High-Level Emulation (HLE), which minimizes both invocation frequency and translation cost by matching and replacing patterns as smaller ‘idioms’ [84] or as larger ‘functions’ with hand-tuned logical equivalents in the code of the machine space.

What is troublesome or ironic in this regard is that when encountering performance problems with a certain virtualization, the tendency is to sidestep the separation entirely. For instance, in the case of programs contained in operating system processes that need to exchange data, the common mechanisms using monitored barriers, such as sockets and pipes, might turn out to be too slow or intrusive due to rescheduling forced by a context switch. This may well be desired from the perspective of an operating system, but could be devastating for a certain process. The compromise is to use shared memory where some memory pages are mapped to belong to several processes rather than reserved for one. This reduces the benefits of the separation as a resilience mechanisms, but at the same time increases the complexity of any analysis activities, as an analyst debugging the program has to take the likelihood of shared memory pages into account\footnote{These contents can also act as external state-holders as per the model in (Fig. 5.5).}. Another way to illustrate the problem would be in terms of inter-
5. USE AND MISUSE OF VIRTUALIZATION

5.4.4 Homogeneity

An interesting effect of whole-system virtualization, is the initial homogeneity that can follow. With homogeneity we mean the extent that the code in a virtual space can be executed on a wider assortment of underlying machines. This is possible if the virtualization can achieve the necessary translation, and may thus need to be extended or ported to fit different machines. Thus, to a certain extent, homogeneity is a desired property. The larger problem however, has to do with retaining homogeneity as time goes by.

The main advantages that can easily be associated with the homogeneity caveat is partly compatibility and partly maintenance. If we know that the number of software instances and devices being maintained are identical, or at least behave similarly to a large extent, planning, deploying and verifying updates and improvements is an easier task compared to the case when all administered components have unique properties to take into account. The compatibility situation is similar. Considering the costs involved in developing software, it is often desirable to reach as many customers as possible.

![Figure 5.8: The feature set of two machines (a and b), and the feature-set (c) of an abstract machine that a virtualization implements.](image)

On the other hand, with the rich assortment of machines and environments that are in play and may be indirectly targeted, the challenge quickly becomes to demarcate the software to some degree by specifying which environments that are to be supported. With respect to virtualization, this means that the situation depicted in Fig. 5.8 can happen, and no matter how the code in the virtual space is formed, there will still be a relative complement of features that cannot be reached directly. When a developer that, for whatever reasons, needs to reach such features the available options are limited: He or she can either remove the virtualization and maintain different versions of the code that used to be in the virtual space or widen the virtualization to encompass all desired features, and many
developers opt for the latter solution. This is achieved through something called ‘native interface’ (like the JNI facility of Java [92]) or ‘local bindings’. Setting aside all the pitfalls involved in such an effort, the net effect is that many of the virtualization benefits have been set aside and will be hard to regain.

5.4.5 Compatibility

Using virtualization as a means for establishing compatibility can, in a sense, be achieved by retaining homogeneity. If the code in the virtual space is compatible with other machines, the virtualization itself should, of course, be capable of execution on said machine. A special case of this scenario is the briefly mentioned backwards compatibility. This compatibility is backwards in regards to machines that are no longer actively produced, accessible or in any other way not worthwhile to maintain in their original shape, but for which there still is interest and demand for allowing previously written code to execute on more current machines. This may be done for a variety of reasons. For instance, reimplementation of the code may be to costly or otherwise infeasible, or the reason may be historic preservation.

Even if it evidently is possible to accurately reproduce or mimic behaviors of a machine different than the one currently executing, it is a far greater challenge to guarantee equivalence in every sense of the word (computation performed, costs in terms of time and space). Thus, this is a risky subject for dependable computing. The primary reason is the rich assortment of conventions and protocols in play and the lack of available specifications of said conventions and protocols [35]. Take, for instance, the MOS-6502, which was arguably one of the more well known and used processors during the infancy of personal computers. There are numerous virtualizations that can interpret these instructions as part of their virtual space, but on close inspection, they yield considerably different behaviors, particularly in regards to timing (cycle accuracy), bugs and undocumented instructions that a lot of existing code takes advantage of. It is not until very recently [36] that there is an accurate description for one or a few versions of this processor against which to verify emulator behavior.

Adaptive Optimization

When using adaptive optimization techniques, homogeneity during execution is purposefully ignored and considered only a static artifact. Each virtual space is actively re-tailored to match the current state of affairs and the current capabilities of the machine. In theory, this has the potential to beat static compiler optimizations, because of the gained ability to dynamically, rather than predictively, manage changes to the conditions of the machine including instruction scheduling and throttling CPUs. In practice, since the only ever acceptable code alteration that should be performed is the one that gives net gains in performance, many optimizations will be infeasible as adverse effects to shared external state holders are difficult to establish in advance.

In principle, the overhead in the dynamic reevaluation of code and of monitoring changes to operating conditions is accepted to gain the possible advantages of
5. USE AND MISUSE OF VIRTUALIZATION

code more tuned to a specific machine. It is true that the overhead involved can be made comparably small, that there can be large gains, and that the overhead is to some extent necessary in order to combat other performance degradations as described in Sect. 5.4.3. On the other hand, these optimizations necessarily involve quite drastic dynamic alterations to the code in the virtual space or to its translation, sometimes to an extent that there is interference with other goals. These goals include security concerns such as interference with code-signing, antiviral pattern matching, dynamic protections like WX [117] and opening possible new side-channels for timing attacks on cryptography – all of which may have severe consequences.

Another serious concern is the ever-present possibility that the altered code may trigger some unexpected corner case and influence the computation adversely. Compiler bugs are far from unheard of and a dynamic optimizer shares some of the same risks. Bugs introduced at such a stage have all the hallmarks of being difficult and expensive to resolve, especially since the criteria that triggered optimization are not stored and are not likely to be contained or deducible from a snapshot of the virtual space.

5.4.6 Software Security

Software security is troublesome in many ways, partly because it is usually sufficient with a known, exploitable problem, i.e. a vulnerability, for a software to go from supposedly secure to guaranteed insecure. Starting with the common problem of homogeneity, we will now discuss the darker side of the write once-run everywhere mantra. The problems related to this can apply to programs that the stakeholders would not want running but which may still be part of the antagonist’s attack strategy, e.g. viruses, backdoors, worms, ... Technically advanced attacks on the one hand, like remote-code execution with Return-Oriented Programming (ROP) [37]-based payloads, require a lot of target-specific fine-tuning to successfully take control, and thus represent a smaller attack vector than the one found in a more widely deployed virtual machine. Furthermore, if the virtualization is so large as to contain the antagonist’s direct target, there is less incentive to create a lower-level machine-specific directed attack. This is a major factor behind the recent rise in web-browser directed counterparts: most of the interesting and sensitive data is present and accessible within the virtual space(s) of the browser. A strong example to that effect can be found in the detailed and advanced exploitation of an integer overflow vulnerability in a certain version of the FlashVM [118] a vulnerability that was used to break through several levels of virtualization-based protection.

Additionally, it has been shown disturbingly often how the seemingly strong protection of many virtualizations can be circumvented to expose or directly program the underlying machine(s) and even how several virtual spaces can be coordinated into obtaining privileged control of a target [119]. Sometimes, the virtualization-friendly features of hardware are taken control of in order to virtualize the host Operating System (OS) itself [120]. This can be further fueled by the potential conflict between, on the one hand, interests which seek to protect a certain piece of software and its data from unauthorized access (which is the
5.4. Caveats

case with many copy-protection systems and other forms of surreptitious software [74] and Digital Rights Management (DRM)) and, on the other hand, users with needs within respective legal systems that are hindered or counteracted by these protections. A drastic case in this regard can be found in the case of a certain piece of DRM protection bundled with some music CDs [38] where the DRM system installed itself in a lower ring of protection (as a device driver) instead of as a regular process and, in doing so, exposed privileged features that were otherwise inaccessible or protected. Viruses were soon developed to take advantage of these exposed features.

Lastly, some security measures placed in outer rings of protection do not necessarily propagate inwards, meaning that if some part of the virtual space allows for the same principal security issue that the protections were added to resolve, they may need to be reimplemented in the virtual space as well. As a simple example of this effect, consider tracking ‘cookies’ as a feature in web-browsers used by some websites to track user activity not only on the own site but on other sites as well, essentially recording the user’s browsing habits. To assist the privacy-minded users, some web browsers added the ability for a user to wipe such cookies when the browser terminated (or even reject them from the start). However, as this tracking technique only requires a bidirectional communication link and some form of persistent storage, it is trivial to re-implement the feature within some nested virtualization that is unaffected by the browsers ‘wipe’ feature – such as the Local System/Service Operator (LSO) facility of the Flash Virtual Machine (VM).

Concluding this section on virtualization caveats, there are a few major points to emphasize:

1. A problem can inadvertently be reshaped from appearing in one form (such as a terminal conditions in the form of a crash) to instead manifest in another (such as insufficient or degraded performance).

2. Circumventing performance degradation linked to a virtualization risks stripping away some of the benefits of the virtualization, or increasing the complexity of the dynamic behaviors of the end system.

3. Static behavior can inadvertently be changed into dynamic.

4. Virtualizations are likely to leak, meaning that they expose properties of their underlying machinery, breaking encapsulation. This can happen directly through inaccuracies in protocol implementation or, perhaps more likely, or through the use of side-channels in another resource, or through the use of external state-holders such as a foreign clock.

5. When the code executing in the virtual space depends on the behavior linked to leaked properties or external state holders, virtualization benefits may no longer apply.

6. Benefits from one level of virtualization are far from guaranteed to be inherited when chaining several together into hierarchies of nested virtualizations (See Fig. 5.12 for an example).
5. USE AND MISUSE OF VIRTUALIZATION

Figure 5.9: At each step, we have general activities that refine our view of the system or physically change it.

5.5 Moving Forward

If we take the aforementioned caveats into account, and assume that we need virtualizations on many different levels in order to avoid reducing computing back into the realm of automated calculation, it becomes necessary to refine our use of virtualizations, and increasingly integrate them into serious development and maintenance processes. In this section, we explore a series of principles that, to a varying degree of system specificity, aim to accomplish such goals.

5.5.1 Prerequisites

For the following principles to be applied and evaluated against a specific target, an experimental environment that effectively encapsulates a subject is needed. This environment acts, in a sense, as a virtualization itself. Such an environment must provide a setting in which the effect of the generic actions that are depicted in (Fig. 5.9) can be evaluated. We have previously worked on such environments for the evaluation of infrastructure protection measures [39] and are currently working on refining these environments [4]. Hence, an environment sufficient for experimenting, applying and evaluating these hardening principles to a software-centric subject, follows the same rules and limitations as other virtualization. In other words, the phenomenon is recursive.

Methodically speaking, the actual process of working with these principles on the experiment environment and on a subject, is similar to that of other experiment-oriented endeavors and follows the basis laid out in [85]. The starting point is an initial view of the system, (Fig. 5.10) which refers to the analyst’s current perspective of the system. From here, one out of several generic actions can be carried out, (Fig.5.9). These actions are: subdivide, measure, intervene and represent. Each of these actions aims to advance the analyst’s understanding of some phenomena in a system to and beyond the point where he or she can take remedial action.

To examine the actions a bit more closely:

- To subdivide means to traverse up or down the chain of virtualizations, consequently widening or shrinking the scope of the (sub-)system studied.
5.5. Moving Forward

Figure 5.10: An interactive, explorative process. Starting from an initial view of the system (v0), we progressively achieve a refined understanding of its constituents and their respective behaviors.

- To **measure** means to gather data using whatever means possible, e.g. logging devices, debuggers, data probes, system traces.

- To **represent** means to take the measurements and transform them into a more intelligible form, either as **native** representations such as the source-code mapping done by a debugger on a triggered breakpoint, or **non-native** representations (Fig. 5.11).

- To **intervene** means to alter or tamper with the subject in some way, e.g. fault-injection.

What is currently missed and underdeveloped here, but relevant considering the scenarios depicted in Sect. 5.1, is the underlying assumption that a virtualization ideal can be established at some level. That level can thereafter be considered the current experiment environment and that the activity of subdividing into secondary and tertiary etc. levels rely on this assumption. However, as there are many scenarios where this is unfeasible and where the opposite will be true, i.e. where there is limited control with an unknown number of central components and where the level of virtualization used as experiment environment may need to link with similar or related environments such as the internet. Preliminary work to that effect can be found in [4].
5. USE AND MISUSE OF VIRTUALIZATION

Figure 5.11: Two examples of non-native representations used to hi-light specific attributes. To the left, there is a poorly balanced binary tree and to the right, some mild fragmentation in a memory or file system.

5.5.2 Principle One, *Tighten Boundaries*

The first principle concerns the circumference (horizontal) and level (vertical) of the intended virtualization. To better illustrate the underlying intent, consider (Fig. 5.12) which depicts a non-exhaustive map of some of the abstractions probably involved in running a piece of end-user intended software on a reasonably modern computer, roughly ordered by level of abstraction compared to the physical machine. Note that there is a certain amount of overlap that is not being shown clearly, particularly horizontal boundaries on higher levels as some abstractions, e.g. streams that can map to either files, communication sockets or processing like encryption or compression, dilute the distinction between primary resources\(^\text{12}\). Also note that the proper function of a given position depends on the proper behavior of all underlying levels and that these levels also correspond to several individual dynamic adaptive systems.

\(^\text{12}\)As an exercise left to the reader, try and construct a similar, but more detailed model representative of a complex execution setting such as a web browser complete with plugin systems, parsers for various scripting languages and the myriad of markup languages needed, and assert which of these that are, in fact, necessary and which can be omitted.

![Diagram showing abstractions involved in execution](image-url)
To begin applying this principle, start with an overhead perspective of resources and virtualizations involved in the targeted system, along with interdependencies. Then establish which of these are necessary. This is relative to the overall stage of the system in question, ranging from in development (design, ...) to deployed. In theory, this ought to be more beneficial the earlier in the process the principle is actively applied, but chances are that other decisions, such as the selection of supporting technologies and target environment, will override and take precedence for political reasons, if nothing else. However, it is plausible that the boundaries of the virtualizations employed in a solution will widen as the system ages and new technologies are introduced and new layers are added in an effort to retain compatibility and increase mobility. However, to do so risks introducing redundancies that in time will become integral parts of the system in question.

To illustrate this with a simple scenario, consider a situation where you have a piece of software designed to execute inside a virtual machine, like the Java Virtual Machine (JVM). The virtual machine environment is supposedly extensive to the point that there are but a few conceivable functions that are not already part of the rich API exposed to the virtual space, and that a desired benefit is the compatibility between a variety of platforms. A managerial directive arrives which dictates that the parts of the services that the software is instrumental in providing are to move to an off-shore data-center, to cut costs like energy consumption or system administration. Sometime after migration is completed, the software starts misbehaving. As a quick fix, a snapshot of the original environment is generated, transferred to the data-center and, to combat differences in hardware configuration, executed inside another virtual machine. Quick fixes quickly become permanent. Now, the actual target and the breadth of the system extend way outside the initially intended scope.

A recent example in this regard (which also applies to the second principle, reinforce borders) can be found in web-browser development, where the tendency is to maintain multiple browsing processes (split across different graphical containers like windows or tabs) within one logical operating system process. Due in part to the complexity of the protocols and data formats involved, there are issues related to achieving a separation between these processes in any sense of the word, as well as strong adverse consequences to security, stability and other aspects of the system. The strategy then, is to take advantage of the process model of the underlying operating system and map browsing processes to operating system processes using some configurable demarcation (per site, per container) [40].

To summarize this principle as a set of imperatives:

- Virtualize only the resources that may explicitly benefit from virtualization. The estimation of benefits should contain the overhead of not only the use of said resources, but the added cost of runtime translation and maintenance.

- Take full advantage of the capabilities of existing, current, virtualizations before adding new layers.

- Dynamic, adaptable reconfigurable execution or management of a resource should be a controlled exception, rather than the norm.
5.5.3 Principle Two, Reinforce Borders

The second principle connects to the previously covered notion of protocols and conventions, essentially verifying that interactions intended to be protocols have not degenerated into convention, and, at the same time, detecting and isolating existing ones. The interfaces that exist between the virtual space and the virtualization, as well as between the virtualization and its machine, essentially form logical borders between subcomponents. At these borders, we can add additional evaluation criteria that assert that the incoming / outgoing data conform to any known, accepted patterns and discard or otherwise react to non-conforming ones.

Looking at the C snippet in (Fig.5.13) as an explanatory aid for this principle and some of its problems (it can be noted that while this is not especially helpful in describing the protocol aspect), the interface of the function is fairly obvious, provides a basic understanding of the existing C type model. For arguments which fulfill the illustrated boundary conditions, the additional check does not achieve anything useful during execution. However, had it not been there to stop the flow of non-conforming data, some kind of disruption would have been likely to follow. Yet, some other possible boundary conditions are still not considered (does src converted fit into dst, do they point to properly allocated, accessible and aligned memory addresses and so on). There are fiendishly many details to get right even in a trivial case such as this one, and with a communication protocol there is usually a time component involved as well. This is, however, somewhat contradictory to the rule of thumb in Jon Postel’s well-known quote from RFC 791 (The Internet Protocol), “In general, an implementation must be conservative in its sending behavior, and liberal in its receiving behavior” [93], which, should everyone adhere to it, would essentially provide bidirectionally enforced borders, although only implicitly in one direction. Considering the wealth of internet protocol stack implementations that are not conservative in their sending behavior and even unstable in their receiving behavior, the long-term repercussions for leniency towards non-conforming implementations or outright ambiguous specifications are quite severe.

```c
#define VALID(X) ( (X) > 1 && (X) < 16 )

int cconv(char* dst, int dtype, char* src, int stype){
    if (!dst || !src || !(VALID(stype) && VALID(dtype))
        return -1;
    /* ... */
}
```

Figure 5.13: A small snippet from a text conversion routine.

In any case, the solution is easy to suggest but hard to realize. There are many suggestions on how to further validate the respective inputs and outputs including model contracts, test-driven development, theorem solvers and other techniques. Some of these operate from the perspective of a programmer actively working on developing a system, having the tool chain essentially refuse to output a binary that does not comply with validation requirements. The focus here,
however, is systemic and gives run-time little or no access in regards to the artifacts used in developing the system. In addition to this, there are both modern and legacy components that, in spite of efforts to the contrary, may for some reason need to be integrated. Furthermore, there are mechanical anomalies (bit-flips, faulty cables and in other ways decaying hardware) that still need to be accounted for and dealt with to avoid propagating silent corruption and similar problems that plague many systems [41] [42]. There is thus incentive to reinforce borders bi-directionally both horizontally and vertically and there are many techniques actively in use (e.g. canary values, data structure checksums, address-space layout randomization, etc.) to safeguard against certain problems occurring after the fact, especially on the dynamic prevention of successful exploitation of vulnerabilities of the buffer overflow type.

5.5.4 Principle Three, Act on Anomalies

With the first two principles, the aim was to establish a sort of foothold from where we can further refine our understanding and control of the subject. There is plenty of leeway in how those principles can be applied. From now on, we can assume that there is some sort of demarcation in place and that it is possible to roughly distinguish between normal information flows and anomalous\textsuperscript{13} ones. The follow up then becomes what to do with the anomalous flows that have been discovered, or, more specifically, which interventions that are reasonable to implement?

One way to approach the situation is through the idea of limiting cascading faults. Using the systemic perspective from Sect. 5.1, we can see that proper execution is dependent on a series of tightly interlinked parts where the desired function of one component to some extent depends on that of the others, and when combined and executed it has non-linear properties due, in part, to feedback loops. Furthermore, from the examples in Sect. 5.4 we find that somehow corrupted or malign computations can pass through many such components undetected, probably increasing the harm caused and making it difficult to determine the underlying causes. To illustrate an approach that may be used to combat this scenario, consider this set of imperative interventions:

- \textit{fail early} – Activate a fault trigger at the earliest possible stage and avoid introducing context switches and other forms of preemption. Also prevent further modification of shared and persistent state holders. Due to the rapid speed at which these systems operate and change state, the window of opportunity where valuable information for determining the underlying cause of the anomaly can be detected is small.

- \textit{fail often} – The fault trigger should, to as large an extent as possible, not be dependent on external states or heuristics. Accessible and proximate sources of disturbances that influence the path execution takes (such as the seed of a pseudorandom number generator) should also be recovered and

\textsuperscript{13}Note that error handling as part of a protocol (error return codes, named exceptions, ...) are not by themselves anomalies, as a protocol dictates what is standard, normal and expected.
5. Use and Misuse of Virtualization

accounted for. Ideally, all conditions are reproducible to the point that the underlying cause can be triggered with as small a delay between reproducing the conditions and reactivating the trigger as possible. However, such an ideal requires that the virtualization ideal holds true for a certain anomaly.

- **fail hard** – Finally, there should be a chain of command and responsibility associated with the fault trigger, meaning that the event is not simply tucked away in an event log somewhere. Instead, data gathered should have a clear recipient that has the ability and mandate to act upon such events. This might be a sole administrator, a tiger team within the organization or even another program, like an Intrusion Detection System (IDS).

This is indeed very similar to approaches found elsewhere, such as **fail-fast** [43]. Note that the recurring notion of a fault trigger does not necessarily have to be something as crude as a program crash or a watchdog initiated reboot, even though these are cases that provide strong data on the events occurring immediately prior to an undesired event. Also note that the interventions outlined above are intended as suggestions and examples of the principle as such, and that the most suitable ones will vary with the system and context at hand.

#### 5.5.5 Principle four, Implement Monitoring

The fourth and final principle concerns monitoring. Consider the following scenario:

We have a background process (daemon) that enables some system maintenance services. Some kind of software bug triggers a starvation situation, causing the daemon to get stuck in an infinite loop on some evaluation criterion that can never be fulfilled. This is a common enough problem where execution is still performed but the overall computing cannot progress. Without extensively examining the executing code (and even then there can be external state-holders preventing the possibility of determining this algorithmically), the process appears – from a systems perspective – to be actively running, which it is. The consequence is that available resources will be eaten by the daemon (hence ‘starvation’) and this process will continue until some external intervention breaks the cycle. Fortunately, most operating systems provide the possibility to coarsely monitor processes (ps, top, activity monitor, etc.) for reasons such as the one described, allowing an operator to make informed decisions. Had this ability been the excluded, the chances for early detection of such performance degradation would have been much smaller and the system may well drift into problems with more severe consequences.

The idea is thus that with dynamic system behaviors follow the considerable risk that previous perception can be invalidated and that this might go by unnoticed. Thus, to avoid the case where decisions and interventions are based on a faulty premise, key state transitions have to be monitored. Unsurprisingly, the level at which this is performed, or which variables that are used for monitoring, is context sensitive and dependent on the system at hand.
The important thing is that decisions are based on the benefits (Sect. 5.3) that are desired in respect to the related caveats. This is not itself without challenges or drawbacks, and to address these (in respect to an operator or other stakeholder) it is important to:

- Collate the variables in the form of representations (monitoring models) that are accessible and comprehensible to the stakeholder.
- Ascertain that the information contained in the representations is sufficiently detailed to empower the stakeholder.
- Establish which reactions that should follow (Reinforce principle three).
- Provide training tools and scenarios tailored to the model(s) and desired reactions.
- Validate the monitoring against the system at hand.
- Keeping the monitoring in synch with changes to the overall system.
- If security is included among the benefits, the monitoring system must also be included, as it operates from a privileged position and is thus a likely target for attack.

5.6 Concluding Remarks

In this chapter, we have argued in favor of a very broad view of the problems associated with virtualizations in a variety of forms, and we have discussed how these need to be controlled and managed in order to iteratively harden and improve software-intensive systems towards a more dependable and secure state. We have furthermore shown how this view relates to probable expectations on positive returns from virtualizing one or several resources, and some of the more daunting consequences that may follow their mismanagement. At the same time, we have emphasized the role of controlled virtualization in order to further computing. Lastly, we have suggested a series of steps and governing principles that, when properly employed, should combat the majority of issues described.
6 Experimenting with Infrastructures

The layout of this chapter is as follows:
In **Background** we provide problem descriptions, background and history relevant to this work, combined with a brief summary of previous efforts. Thereafter, **Experimenting with Power grids** details the design of the power grid laboratory environment and some of the challenges involved when experimenting within that particular domain. This is followed by a corresponding section called **Experimenting with ICT** that describes our general approach to creating robust software environments supporting controlled experimentation. The main section, **Experimenting with Power grids and ICT**, combines these two environments into a unified experiment environment where we can study the interfacing between different critical infrastructures. In **challenges** we briefly describe some issues, both open and closed, that appeared while establishing this environment. Finally, in **opportunities** we elaborate on some of the possibilities that this particular setup provides.

6.1 Background

The need for strong experimentation, verification and validation efforts able to transcend traditional infrastructural and scientific borders is great. If we are ever to successfully restructure and improve present critical infrastructures to fit and surpass current and upcoming challenges, the settings that enable such efforts will need to undergo a similar enchantment, both in a macroscopic and a microscopic sense. With a practical focus in this regard, this chapter looks at the interfacing between infrastructures at two similar, but distinct, levels. The first level concerns the interfacing between power grids and ICT, and the other regards the interfacing between the experiment environments of these infrastructures. The underlying motivation is partly fueled by the european FP-6 project Integrated ICT-platform based Distributed Control in Electricity Grids (INTEGRAL) [105] and the SEESGEN-ICT thematic network [106]. The INTEGRAL case actually covers the integration and interfacing of three different critical infrastructures, i.e. the electric grid including renewables, a customer-oriented business process infrastructure and a SCADA – ICT infrastructure. The SCADA – ICT case is here of particular interest as we discuss ways of improving resilience through the implementation of self-healing mechanisms as a response to some harmful or even catastrophic event. This is especially relevant because of the inherent coupling between the SCADA and the grid – and because any event affecting the grid in such a way that some form of remedial action is needed, may also adversely affect the ICT support needed to perform such actions.
When we cover the design of an experiment environment for a microgrid–ICT setting and a different one for an ICT–ICT setting, there is a bias towards the ICT–ICT problem since in this context, ICT and its actual role is the least understood, but also because of the recursion involved, i.e. the need for ICT as a means for observing and intervening with ICT. In terms of related work or other approaches, the environment presented in Sect. 6.3 has a similar idea and similar goals in terms of overall architecture as the NSF GENI [111] efforts (Fig. 6.1). A contrasting frame of reference between the work in this thesis and in GENI may thus be found through planetlab [44], but this mainly concerns scale and application domain specificity.

6.2 Experimenting with Power Grids

This section describes the setup behind an experimental environment for, amongst other things, self-healing microgrids as per the description in the previous section. This setup serves as a baseline and fundamental environment that the other sections of this chapter will expand upon. It covers three distinct parts: The physical distribution network, the agent logic and sensing equipment that enable self-healing, and finally the SCADA system itself.

The analog micro distribution network, illustrated in Fig. 6.2, was sized by aggregating some electrical nodes of a real distribution network having 30 MVolt Ampere (VA) of rated power at 20 kV. In order to best represent the behavior of a real network for many RTUs, and satisfy budgetary restraints, the network of a test bench of 30kVA, 0.4kV was adopted. The scale reduction of the microgrid components was carried out through different ratios (power, inertia and voltage) to assure that the system was similar to the real system in terms of the internal static and dynamic behaviors of the network, including aggregated loads, network components (on-load tap changers for instance) and DER/RES. A strong requirement is that the performance of the control systems must be unchanged in comparison with the real system.
6.2. Experimenting with Power Grids

This network has 14 nodes, 17 lines, 10 loads and 6 RES/DER, divided into several areas which represent different network characteristics. On top of this network, there is a considerable supporting ICT infrastructure illustrated in the overview in Fig. 6.3.

A fault location algorithm had previously been developed using MATLAB, and, subsequently, the governing agent was also built using a combination of MATLAB [121] and MATLAB (OPC) toolbox [122]. The OPC Data Access standard over Ethernet provides for a common protocol for communication between the necessary OPC server associated with RTUs (which use TCP [94]/ (Modbus) [95]) and between the SCADA control center and the OPC client (MATLAB OPC Toolbox).

Communicating RTUs such as the Fault-Recorders (FRs) emulators (developed within LABVIEW [123]) as well as the Fault Passage Indicators (FPIs)\(^1\) that are directly connected to computers that are then further interconnected across several local-area networks by the application level communication supplied by the OPC client/server solution. Finally, the advanced control and batch execution used to accomplish the self-healing functionalities can be carried out either directly by the local agents or indirectly by commands issued by a DSO, that is empowered from the information supplied by the agents.

The composition of the OPC real-time communication system for the INTEGRAL demonstrator is shown in Fig. 6.3 and the computation that regulates the system comes from the OPC Client Toolbox and the MATLAB local agent with its embedded fault location algorithm. In terms of data acquisition, numerous current and voltage sensors have been implemented. The dynamic data on distribution network behavior gathered from these sensors are continuously exchanged with the Intelligent Electronic Devices (IEDs), here limited to just the Flair 200Cs and

\(^{1}\)Flair 200C, fault passage indicators furnished by Schneider Electric Telemecanique.
6. EXPERIMENTING WITH INFRASTRUCTURES

LABVIEW Fault Recorders. These are dedicated to remote monitoring of middle to low voltage substations. When the passage of a fault is detected, the current and voltage data are recorded in real time and transmitted to the MATLAB agents via the OPC server\(^2\). The data exchange is then performed between the OPC server and the MATLAB OPC client toolbox. Afterwards, the exact position of the faulty segment will be determined by the fault location algorithm. The computed action, i.e. which circuit breakers or switches that are to be opened or closed in order to restore as much load as quickly as possible after the fault has been identified, is then relayed from MATLAB to the protection and automated devices in the distribution network via the SCADA system.

![Diagram of the ICT solution](image)

Figure 6.3: Early overview of the ICT solution. Note that parts of the base ICT layer are not covered by corresponding monitoring nodes.

6.2.1 Communication between IEDs and SCADA Software

In order to supervise, control and communicate with each and every automatic device and software, the supervisory software PCVue [124] is used\(^3\), allowing for the support of a very wide range of industrial SCADA protocols. The communication between the supervisor and the process equipment, is handled by a component called the Supervisor Communication Manager. The communication technologies involved include OPC, the native driver equipment protocol, DDE [96], etc.

In the self-healing demonstration, the OPC technology is used to communicate between supervisor and local intelligent agent, while the native driver equipment protocol is used to link the industrial Programmable Logic Controllers (PLCs).

---

\(^2\)For this purpose, a Schneider Electric OPC server product called OPC Factory Server (OFS) is used.

\(^3\)PCVue is a SCADA solution for multi-station supervision and control, developed based on the considerable industrial automation sector of the ARC Informatique Company, and as such is representative of current state-of-the-art in SCADA.
The equipment devices are labeled nodes of a particular network. The messages that pass between the supervisor and the process equipment are called frames. The mechanisms and boundaries which allow the supervisor to interact with the MATLAB Intelligent Agent and the industrial PLCs are the following:

- 16 simultaneous communication channels, each with their own protocol.
- The refresh-rate is fixed to the rate specified by the frame-scan rate.
- A real-time kernel between the supervisor and the process equipment, periodically refreshes the values of variables in the Supervisor database, using data from communication frames.
- Each entry in the Supervisor database that corresponds to an equipment source is further linked to a specific location in the communication frames. At least one corresponding entry has to exist for a link to be established between a frame and the database.

6.3 Experimenting with ICT

There are several deeply rooted issues when experimenting with ICT and many of them stem from the extremely heterogeneous and dynamic nature inherent in software-intensive systems. This section will briefly discuss some of the generic, overarching problems that need to be taken into account when using and seriously experimenting with software. We will thereafter describe the overall concepts of a system that can be used to generate and maintain software based experiment environments able to strengthen control and improve the accuracy of the gathered data.

Modern software is strongly structured around various hierarchical forms of separations that are tied to some abstraction or to the semantics of surrounding systems. Some of these are enforced by the technology that makes software run, execute, as part of performing computations. Others are simply modeled in order to, in some way, assist the development of software. There is, however, a certain degree of overlap between the two. For the sake of reference, the abstractions that are strictly modeled are here called static and can as such be studied and processed by tools and methods other than computers. Many such separations are simply stripped away or reconstructed, optimized, into something more efficient during the translation from human-readable formats, source-code, into a format efficient for computer execution, i.e. binary-code. Furthermore, the effect these static abstractions may have on program behavior can, to a fair degree, be predicted\(^4\) and determined as benign or as undesired in advance. This can be done using formal techniques such as model checking and theorem solvers, or through simulations on a modeled machine. An often held fallacy in this regard, however, is that the source code used to construct a software system is strongly representative of the program(s) that will execute on a computer [28]. For the

\(^4\)Even so, the limitations that stem from the age-old computability ‘halting problem’ still apply.
other category, *dynamic abstractions and separations*, observable computation patterns, i.e. behaviors, are by and large undeterminable up until essentially the point where their corresponding execution is performed. This is because executing software exhibits advanced characteristics such as *non-locality*, *heterogeneity*, *recursiveness*, *polymorphism*, *re-connectivity* and *concurrency* in addition to a very large space of possible states – all of which are influenced by the information the system receives, processes and transmits.

The challenges of software make it tempting to break it down into smaller chunks (programs, objects, libraries, processes, threads, etc.) and strategies are employed both statically and dynamically to enforce and assure the separation of these chunks. The dynamically enforced separations are referred to as *virtualizations* and can be found at a variety of levels of granularity among which the more commonly known one is the notion of *processes* in modern operating systems. The separation provided even in those cases is, however, to a varying degree, insufficient to safe-guard against all the aforementioned execution characteristics. As shown by Fig. 6.4, the full execution of a bounded computation, virtualization, is ideally dictated solely by its initial configuration (its code and its input). However, there are dynamic sources of information that are necessarily external to the virtualized space which still influence the execution in such a way that the protective enclosure can be breached (means to intentionally do this is a currently an active area of research within software security). Thus, even though the intended target for experimentation is encapsulated using some form of virtualization, an important step in initial experimentation is to isolate and control such state-holders.

Means for performing controlled experimentation on fairly strong virtualizations, such as processes which confine programs, are well developed and numerous. However, when the form is less traditional, which is arguably the case with critical infrastructures, the tools are far less advanced.

---

5In fact, they do not even strictly have to be a part of the software-system in general, but may as well come from the surrounding information system or from a user.

6This is a rather brief and shallow summary of the problem. The full extent of this discussion is,
As an approach to expanding this control to incrementally larger borders, a solution called EXP [39], was developed as part of previous projects [104]. The Borders of EXP is illustrated by Fig. 6.5. EXP is partly a hierarchical data-model describing abstract roles that can then be assigned to lab-nodes in the environment, and partly a set of services that enforce the policies defined by the roles of the nodes when combined. The major services, as shown in Fig. 6.6 are thus:

- **Startup** - Role-specific bootloader sent over the network to affected nodes to control node startup, used to run integrity/hardware checks, as an enabler for other services and to activate the current configuration in a controlled manner.

- **Restoration** - Provides the ability to generate snapshots of the inactive state of a node but also the ability to revert to previous snapshots.

- **Experimentation** - Miniature, low-footprint FreeBSD-based OS for quickly deploying small agents to act as input or noise (network traffic generators and the likes) to main nodes.

however, well outside the scope of this chapter.
In addition to the nodes used for experiments, there is also an additional node reserved for coordinating the others. This node is called a **controller**. The responsibilities for providing and managing services and nodes are primarily put on the controller. This enables two distinct modes of operation: **Deployed environment** and **Sustained environment**. In a deployed environment, the nodes have physically been hooked up to the controller for configuration. When configuration has been completed for all involved nodes, the controller is either disconnected from the network or reverted into acting merely as a network router. By contrast, in a sustained environment, the controller actively micromanages the nodes involved as well as acts as intermediate storage.

Expanding on these ideas, the ICT part of the environment detailed here takes two geographically separated, sustained EXP environments and combines them into one larger environment. This distributed environment is called EXP-II and can alternate between a distributed and an isolated setting as well as provide monitoring services using dynamic tracing combined with post-mortem analysis, for both online and offline data-acquisition on ongoing or completed experiments.

![The basic lab setup.](image)

The initial structure for the EXP-II environment is depicted in Fig. 6.7 and corresponds to an EXP controller, an IEEE 802.1Q (VLAN) [97] capable switch and three lab nodes that each manage a subnet, as shown in Fig. 6.3. The quality of the nodes and their parts were on the level of Commercial, Off-the-Self (COTS). Two such setups were created, one for each of the two geographical locations. This model will be expanded upon in the next section.

---

7In some cases this includes their power supply using programmable outlets.
6.4 Experimenting with Power Grids and ICT

By taking advantage of both experiment environments as detailed in the previous sections, the task becomes to combine the two into a unified experiment environments. This must be done while still maintaining the respective benefits, reductions and structure of the individual environments but at the same time open up for new opportunities without compromising functionality, integrity or security.

![Diagram of the basic lab instantiated, one mode of operation.]

6.4.1 Isolated Operation

To bootstrap the experimentation endeavor, an incremental approach was ultimately chosen. The first step was to define and construct a software configuration capable of virtualizing the communication between three larger slices of the SCADA system in the microgrid. To this end, the abstract experiment environment detailed in Fig. 6.7 was instantiated as shown in Fig. 6.8. The software configuration for the three nodes was running the FreeBSD [125] operating system, configured as simple dummynet [45] routers.

The dummynet configuration was added to be able to emulate a variety of communication links, and it could be changed at will by an operator. A major criterion for this stage was that the environment should operate as an isolated cell, without access to external communication through a corporate Wide Area Network (WAN) on the internet.

When the environment is in an active state, meaning that the experiment or demonstration is running, the controller also routes between the nodes and their subnets, aggregates the acquired data and sends out probe signals at frequent intervals to verify and track current dummynet configuration. The actual data acquisition occurs through raw packet recordings of each individual node on the interface connected to the monitored subnet.
6. EXPERIMENTING WITH INFRASTRUCTURES

6.4.2 Joint Operation

The second step, then, was to establish the environment in a joint-operation mode. In this mode, we have a situation where two controllers are connected to an external, possibly hostile, environment; making security concerns more pressing. To deal with this situation, and to establish a trusted connection, the respective firewalls by default refuse all incoming traffic. Then, at agreed upon points in time, the remote location opens up for a single incoming connection from a fixed source address. The other location initiates the connection through which a Virtual Private Network (VPN) tunnel is established, using pre-shared keys that have been exchanged offline. The difference in terms of information flow is that the traffic between the different subnets is now duplicated and forwarded to the remote network. Dummynets can still be used, but the default setting here is to have them disabled.

6.4.3 Unified Operation

The third step was to establish the environment in a unified-operation mode, as shown by Fig. 6.9. This mode builds upon the joint operation mode but with several major changes. For starters, the bridging nodes at the local site no longer perform any data-acquisition or traffic shaping (dummynets). Secondly, when the VPN tunnel gets established, the static routes that previously joined the three nodes together at the local site, are altered in order to redirect traffic to take a ‘detour’ through their respective analogs in the remote lab. For instance; traffic going from a FPI through (site a, node 1) destined for the OPC network, will follow the path:

FPI → (site a, node 1) → (site a, controller) → (site b, controller) → (site b, node 2) → (site b, controller) → (site a, controller) → (site a, node 2).

The return-path follows the same general pattern.

In closing, the overall principle behind these three modes of operation mimics some of the ideas powering the microgrids as well. During ideal conditions (unified operation), information is traded between cells (here represented by the two physically separate environments) in a seemingly coupled fashion. Should some event disturb or threaten this setting, the system reverts to a safer mode of operation (joint operation) and should things deteriorate even further, they can be switched to isolated operation. While not currently taken advantage of, this could be an interesting avenue to explore further down the line.

6.5 Challenges

The purpose of this section is to highlight the less obvious challenges involved in constructing the environment depicted in Sect. 6.4, and also to discuss issues that, for practical reasons, were left open or that may prove relevant to projects facing similar problems.
6.5. Challenges

Constructing the unified environment proved challenging in several respects. When the separate problems of experimenting with power grids and with ICT are accounted for, the obvious and challenging part is the inherent coupling between the power grid and its managerial ICT and the here assumed brittleness of the devices and protocols involved in the SCADA process. Since the risk for harming equipment through misconfigured ICT has previously been shown to be considerable [126, 127], and since the nature of software behavior is volatile\(^8\), there are technical barriers which – until cleared through validation, implementing or hardening of safe-guards on border conditions – put heavy restrictions not only on the experiments themselves, but on software and network security as well.

Furthermore, the implicit maintenance requirements regarding the equipment that comprises the physical layer(s) of the experiment environment bring forth additional complexities that ascertain that the two environments have a synchronized configuration to as fine a granularity as possible. One such issue concerned a race condition through the use of a Keyboard, Video and Mouse (KVM) switch to alternate between active nodes when working on several components in an intermittent fashion. As per the principal problem of software experimentation in a para-virtualized setting illustrated by Fig. 6.4, the activities of a KVM is merely one such external state-holder that cannot easily be precisely controlled or manipulated.

---

\(^8\)More specifically, far from all processing circuitry have clearly defined and tested reactions to arguments received as part of communication protocols.
6. EXPERIMENTING WITH INFRASTRUCTURES

6.5.2 Economical Barriers

The ICT experiment coordination facility as depicted in Sect. 6.3 only has managerial control over the three routing nodes in the main lab, along with the corresponding devices in the analogous lab, i.e. it cannot directly roll-back or otherwise control the components outside this abstract perimeter. While the perimeter can be expanded using more specialized technology and software, this has not yet happened and it is not an immediate goal. For this to happen, we speculate that the reasonable, incremental enhancement would first be to improve the granularity of the monitoring to cover not only communication between fairly rigid interfaces, but also the comparably dynamic and adaptive interactions inside the SCADA of the grid-level ICT.

Another strong economical barrier with technical undertones is the cost of maintaining two instances of the same ICT laboratory environment. This is problematic for primarily two reasons. The first reason is that domain expertise may be geographically tied to one (in a collaborative project such as the one depicted herein) or none of the geographical instances (outsourced). Either situation will produce a certain overhead in response times during troubleshooting and maintenance. The second reason concerns the comparably short longevity of core components such as hard-drives [41] that are susceptible to stress. While cheap and easily replaced, such maintenance again needs to be synchronized between the sites and considerations taken of possibly influential entropy (such as wear-leveling in flash-based storage devices) [42].

6.5.3 Political Barriers

Due in part to the incremental development of the distinct (Sect. 6.3, Sect. 6.2) environments, there is an additional dependence in the case of the intermediate ICT infrastructures of the respective local area network environments at both ends. Although virtually separated through technologies such as Virtual Local Area Network (VLAN) tagging, the environments rest on the preexisting networking infrastructure, including Local Area Network (LAN)/WAN border management such as firewalls and intrusion detection systems. When, as in our case, the larger WAN is the Internet, border management policies tend to be very strict and it is therefore likely that incompatibilities arise between such policies when the networks and policies have been developed independently of each other.

Ultimately, this can be construed as a political barrier that causes significant technical consequences. For the particular scenario described here, the major limitation was the need to essentially tunnel TCP over TCP rather than TCP over User Datagram Protocol (UDP) [98] for the VPN. This implies undesired interactions when TCP’s retransmission algorithms are applied recursively; a problem likely to result in a noticeable decrease in network performance. The extent of this problem is lessened, however, by the distinction between isolated operation and unified operation, since the different modes complement each other.

---

9The data being communicated can, of course, be tampered with, but such interventions are comparably coarse in comparison to being able to directly modify the software. 
6.6 Opportunities

In addition to the benefits than can be reaped in the respective domains from the individual infrastructure environments, there are quite a few interesting opportunities that open up when combining the structure from Sect. 6.4 with the considerations in Sect. 6.5. This section will elaborate on a few of the opportunities that may be of interest in the near future.

6.6.1 Protocol Design and Evaluation

There are lots of protocols involved in current SCADA systems, ranging from small, narrow and product specific to broad and generic. Given the fairly radical suggestions regarding the future of the grid(s), there are justified concerns regarding how some of these protocols will behave in new settings, which restrictions they impose on supporting communication and processing technologies, exactly which information is communicated and similar aspects. The datasets that will be obtained from isolated operation can be used to specialize or generalize available protocols to fit new grid structures such as the microgrid, but also function as input to the design of future network protocols and to the configuration of security equipment such as firewalls and intrusion detection systems.

6.6.2 ICT Monitoring Models

The complex interactions between information processing systems, information systems and the physical grid have been a source of much commotion, and one does not need to look further than the northeastern blackout of 2003 for strong examples to that effect. The work involved in discovering the underlying causes behind such events is further complicated by the complexity of software analysis and debugging. With more adaptive ICT systems, more modern network structures and more dynamic services models, there will be an even stronger incentive to monitor not only these different layers individually, as is currently done, but to monitor them in relation to each other. The setup that has been covered here may serve as a useful starting point for the development of such monitoring models, technologies and methods that essentially allow the SCADA concept to be applied recursively, i.e. SCADA for SCADA.

6.6.3 Rogue SCADA

That SCADA systems have a dodgy past in terms of security and notable recent incidents [101,102], have served to emphasize this fact, and it seems unlikely that we somehow will be able to retrofit major developments in software and information security to be usable in current closed and legacy-rich SCADA systems. The structure proposed here allows for an evaluation of the consequences of various kinds of directed attacks and their corresponding protections from the perspective of an antagonist outside the system (denial of service, side-channels of data-flows, etc.). When using the unified mode the proposed structure also make it possible to evaluate what the effects of a compromised cell could be. This work is continued in Chap. 8.4.1.
6.7 Conclusions

In conclusion, tools and means for the controlled experimentation on the interface between ICT and energy transmission in a critical, self-healing context have been introduced. We can use this not only to properly evaluate means for improving resilience, but also to obtain much needed datasets on systemic behavior in several situations characteristic of future smartgrids and microgrids.

To this end, several barriers relevant for those working on similar targets, have also been identified. While the majority of the work described there has been completed or is very near completion, work for the near future involves exploring the opportunities mentioned, but also attempting to generalize this solution to fit other experiment-oriented endeavors and needs from similar domains.
7  Retooling and Securing Systemic Debugging

7.1 Introduction

In its essence, software debugging first concerns the discovery of causal factors (chain of events) behind undesired behaviors and thereafter taking remedial action with the end-goal of preventing said behaviors from reoccurring [46]. The way this plays out in actual software development is, at the very least, two-folded – on the one hand we have the active development or bring-up of an envisioned software-based system (early-stage) where anomalies are uncovered during validation efforts, and on the other we have the reactive response to anomalies experienced by end-users (late-stage). Subsequently, we refer to these two different phases as program debugging and systemic debugging1.

Among the notable differences of relevance to the argumentation herein, is that in the program debugging case, the analyst has access to higher precision and accuracy in measurements, more options for controlling environmental factors and for intervening with the subject. Among numerous non-trivial issues that needs to be accounted for in the systemic case, we find obfuscation, encryption, digital-rights management, unknown third party software (including rootkits and viruses) and the work is often closer in nature to reverse-engineering challenges than it is to programming ones. For these reasons, this chapter will assume the systemic debugging perspective.

As with many other tasks in software development, debugging is not only centered around a set of key enabling tools, but most of these tools are pieces of software themselves – relying on the same mechanisms and theoretical foundations as the systems they are supposed to help fix. Unfortunately, all of the non-trivial issues just mentioned combined with the countermeasures employed to eliminate or hinder these issues also drastically reduce the usefulness of debugging tools overall, to the point that it is worth reassessing the circumstances in which these tools can be useful, and invent new ones to fill whatever gaps we may find.

This article is organized as follows:

In Sect. 7.2, Smartphone Transition we illustrate the breadth of the problem with an industrial case based on a shift from feature phones (i.e. cellphones as static and monolithic embedded systems where the handset manufacturer, network operator and similar stake-holders define the feature set of the phone), into so-called

1For a lengthier non-academic elaboration of the subject, consult [70].
7. RETOOLING AND SECURING SYSTEMIC DEBUGGING

smart-phones (semi-open, heterogeneous and dynamic where the end-user defines most of the software configuration) focused on the configuration, role and interplay of debugging aids from the perspective of a debugging-focused team of specialist (a so called “tiger team”) at Sony Mobile Communications.

Then, in Moving Forward we examine how recent developments in tracing frameworks can be improved upon to remedy the issues that was brought up in the previous section, and then describe the architecture and design of such a tool.

Lastly, in Testbed for Systemic Tracing we introduce the prototype and experimental validation of a free and open tool based on the ideas laid out in Moving Forward.

7.2 Smartphone Transition

Current generation smartphones are interesting from a debugging perspective in that they quite cleanly illustrate an embedded- to open- shift during a compact time-frame. One the one hand, there are older cellphones (“feature phones”) as embedded, albeit quite complicated, kinds of systems that have advanced protocols interfacing the large and legacy-rich phone networks. On the other hand, there are smartphones in the sense of semi-open\(^2\) generic mobile computing platforms where the distinguishing features from other devices such as tablets or net-books are related to form factor, user input methods and other superficial details, rather than the computing that is being performed.

Starting with the feature phone, as representative of a closed but large (millions rather than thousands of lines of source code) monolithic, embedded system; They are closed in the sense that there is limited, regulated or no access for running third party, foreign, code on the device. In addition, third party code, when allowed, is limited to a reduced feature set and only portable across a small span of devices.

The resources that are available to developers are similarly limited and in addition to some real-time requirements on behalf of the communication protocols, optimizations need to consider not processing power as such but rather energy consumption. Furthermore, the memory space of these phones is not guaranteed to be separated between processes. This means that a wild pointer could, for instance, corrupt long chains of code, data and memory-mapped resources belonging to other parts of the system and tight packing of the memory space makes this statistically more likely than in a system where each computing task has a distinct process with a dedicated virtual 32/64-bit memory space. Thus, the execution environment is highly static to the degree that it is trivial to establish a memory map of functions and resources, simply from the output of the static linking stage of a build system.

These properties have interesting repercussions for the kinds of anomalies experienced and how long and far-reaching different kinds of errors can cascade into each other before being detected, often producing severe problems (permanent

\(^2\)Semi-open because these devices are still subject to varying degrees of vendor lock-in along with type approval and other certification processes regulating the rate at which the platform can be updated.
malfunction that results in a 'bricked' device that cannot be returned to a stable state). Furthermore, the high number (millions) of instances of the same system means that even problems with very low repeatability rates will need to be investigated even when the problem in question cannot be directly reproduced.

The smartphone transition involved large changes to the aforementioned points. To begin with, the hardware platform is many times more powerful and, in terms of ability, more similar to that of laptops one or two generations back. The software platforms are available under open-source conditions and third-party developers are encouraged to develop and release their software through built-in, regulated, marketplaces as a means to gain users. Subsequently, the end-user is able to fine-tune and customize the software configuration to their hearts’ desire. However, these changes and associated economic incentives open up for other, darker areas, such as piracy and various forms of privacy invasions. The industry response from affected or concerned developers is, unsurprisingly, to try and protect the software against such piracy through the usual means and obfuscation, DRM and coupling to external state-holders (e.g. web services) are openly encouraged [128]. Such measures, however, further complicates debugging.

The crucial aspect is that the smartphone developers integrate an array of partially underspecified platforms, often without clear service-level agreements. It is really a branded and themed access to these platforms, rather than a feature-specified, controlled and locked one that is being mediated and ultimately sold by the smartphone vendors. Even though the developers behind individual brands have comparably little influence as to the actual software configuration of any specific instance, it is ultimately a notable part of their responsibility to optimize the end-user experience. This calls for both a direct (third party-developer support programs) and indirect (third party software analysis to figure out where and what to optimize) response.

As part of this indirect response, we find the connection between this optimization challenge, the debugging challenge and to other, more subtle ones (e.g. reverse engineering in binary analysis), in that the desired capabilities of the supporting tools are remarkably similar. Therefore, we will continue this section by taking a look at available tools, how they fit together in a larger organization and the requirements that exists which limits their use.

### 7.2.1 Toolchains

Before looking into the specifics of each separate category of tools, it is worthwhile to note that in the setting described herein, debugging tools are never operated in isolation but are rather integrated into the entire value-chain, with intermediate steps being tightly tracked and controlled with users ranging from novices to specialists.

Fig. 7.1 illustrates a drastically simplified such chain, where we can follow how an experienced issue propagate from an end-user (or external voluntary tester with a specialized tracing build) to a service center (performing initial assessment for repair, replacement, snapshotting or special tracking) onwards to specific developer groups within the organization (these include test-departments), and should the
issue be of a more complicated nature, onwards to specialized task forces. *Thus, it is paramount that any new tool or mechanism doesn’t disrupt the flow of such a setup.* Comparing debugging to exploitation, the successful exploitation of software vulnerabilities involve controlling the chain of events preceding an undesired behavior, setting up ideal conditions (weakness) for triggering the behavior and for taking advantage of the aftermath in order to circumvent restrictions (dissolve security properties). As such, there is an inherent conflict between the two activities in that successful debugging implies the removal of a weakness, albeit it should ideally be more difficult to exploit a bug than should ever be to fix it.

Other than that, there’s a surprising amount of overlap in terms of how we figure out the chain of events, and thus, the tools used to assist debugging can also typically be used in developing an exploit. Since dynamic protections are designed to combat the mechanisms used in what would otherwise be successful exploitation of vulnerabilities, relying on the same mechanisms for implementing a debugging tool would imply a forced choice between the one or the other.

*Therefore, the first principal problem for all debugging tools is selecting techniques that would assist debugging but not the development of rootkits and exploits.*

Looking at specific categories of tools employed in an organization such as the one described, we find a few major categories being *symbolic debuggers, tracers, profilers* and *crash-dump analyzers* and will continue this section by highlighting specific issues encountered with each of them.

### 7.2.2 Symbolic Debugger

The arguably most powerful of tools in a software analysts’ toolsuite is the symbolic debugger; a core component of most IDEs, lurking behind features such as breakpoints and stepping through execution one statement at a time.

Breakpoints can, from an abstract perspective, be *any controllable interrupt* (or trap) during code execution that the debugger is able to forcibly introduce (and remove). When an interrupt is triggered, the control is diverted to a handler routine.

---

Figure 7.1: Simplified interactions between tools and organizational units.
inside the debugger. A direct consequence of such unexpected interrupts is that they can invalidate many assumptions made during compilation in regards to instruction scheduling, branch prediction and so on implying that these notably affect the performance of the target even when they are not triggered.

Thus, the symbolic debugger is not a tool that is suitable for the direct investigation of performance degradation, and which should therefore instead be used for protocol mismatches, corrupted data and terminal states.

Part of the utility of a symbolic debugger stem from the ability to use source-code as a form of representation for measurements, and other parts from the ability to not only gather measurements but to actually intervene with the subject and alter its states. In order to achieve this the symbolic debugger needs to be able to exert some level of control of a subjects execution, typically from manipulating code and data in memory with assistance from a designated interface (e.g. POSIX:Ptrace, Win32:CreateRemoteThread, and hardware JTAG). This also makes such interfaces prime candidates for exploitation and all of them have successfully been used in notable exploits in the past. The principal security challenge for the symbolic debugger is thus hardening of debugging interfaces and improving access control.

Other challenges include:

1. The presence and ability of a symbolic debugger can be both detected, circumvented (anti-debugging [47]), misdirected ( [48]) and is further hindered by common security measures such as ASLR [49], DEP/WX [117], etc.

2. Many low-level dynamic states needs to be tracked in order to retain truthfulness [50], e.g. relocations, trampolines, variable-instruction length decoding, state sensitive instruction sets, etc. If a debugger is missing this property, it is not just useless but also misleading.

3. There is a large disparity [28] between the source-code view of the developer, the source-code view of the compiler (preprocessor macro-expansion) and the lower-level code, but it is at the low-level that execution can be instrumented. However, there is no guaranteed ratio between the two, it can essentially be many to many.

4. The debugger kernel has to both track, and be aware, of threading schedulers and other concurrency techniques.

5. The source-code level of representation requires a special build that retains private symbols and, preferably, excludes most or all optimizations.

7.2.3 Tracer

If the symbolic debugger was a narrow category, the category of tracers is far wider. The term tracer refers to all tools that provide some specific traces of execution of a program that are not strictly part of its intended input / output. Subsequently, there is a rich variety in the number of information sources for tracing,
e.g. system logs, process signals etc. These also include the venerable “printf” debug output left behind by careless developers. Furthermore, most symbolic debuggers have some trace functionality added through the call trace, also called stack trace. This means that it will try and extract the sequence of function calls or jumps that led to a triggered breakpoint. This is achieved either by analyzing data on the stack, with varying degrees of difficulty depending on the architecture and on how optimized the code is, or by maintaining a separate stack or log.

Such corner cases aside, tracers are tools with a generic data gathering approach that can take advantage of a wide range of information sources that are not exclusive to dedicated debugging interfaces and other forms of specialized support. In addition, they are not as strictly bound to a single program as symbolic debuggers are.

Some key-pointers in relation to tracing tools:

1. The connection between trace samples and the source of the sample is not always obvious or tracked. Thus, the data need to be tagged with some reference in regards to its source, covering both where (instruction, function, program, etc.) and when (timestamps) the sample was gathered.

2. The imposed overhead varies a lot with the quantity of data and the frequency of samples together with the properties of the interface and the resource that the data comes from.

3. There is no clear or default reference model (e.g. source-code) to appropriately compare and study.

4. As tracing tools are quite easy to develop, you can quickly end up with a large number of unnecessarily specialized tracers, unless the tool-chain is heavily moderated.

### 7.2.4 Profiler

The profiler as a generic category covers performance measurements, but can be viewed as a distinct form of a tracer\(^3\) that specializes in performance degradation problems. The implicit assumption is that the performance degradation is linked to some subsystem where the most execution time is being spent, which is not always the case with, for instance, resource-starvation linked performance degradation.

These measurements can be implemented using two trace sources or a single datasource, a situation which is here referred to as event-driven or sampling-based. With event-driven tracing, there is some sort of reference clock (to establish time or cycles that elapse) and two trace points (entry and exit), such as the function prologue and epilogue in many low-level calling conventions. With sample-based tracing, some data source is sampled at a certain rate, and changes (or the lack of changes) to this data source are recorded. An obvious such source, from a

---

\(^3\)Even though it can be integrated as a part of a specialized build of the software.\]
7.2. Smartphone Transition

low level perspective (particularly in cases where the code distribution in memory is well-known and static), would be the instruction pointer/program counter of a CPU. This even though they may require specialized hardware, but the idea translates to virtual machines as well.

The shared aspect of these tools, however, (perhaps more so in the event-driven case) is that the refined use relies heavily on the analyst’s skills when it comes to statistical data analysis and modeling.

Here follows some key-pointers about profilers (and subsequently, about the use of these tools for debugging performance degradation):

1. Even though the needed precision may vary, the case can, at times, be that the degraded performance is not directly linked to just processing power, but rather to the relationship between different resource types (communication, processing and storage with their respective latencies).

2. When the environment is heterogeneous rather than uniform, it cannot safely be assumed that performance measurements generalize between different instances of the same system.

3. Some scenarios (specialized builds, event-driven tracers) with adaptive algorithms that alter behavior based on estimated processing power (common in audio and video applications), are particularly prone to suffer observer effect from profilers, and the evaluation criteria used by the algorithm implementation may need to be controlled and manipulated.

7.2.5 Crash-Dump Analyzer

The last tool category to be covered here is the crash dump (post-mortem) analyzer on which two perspectives are presented. The first perspective is that crash dump analysis is a specialized part of the preexisting functionality of the symbolic debugger, with the difference being that the option to continue execution is not especially useful. The debugger merely provides the analyst with an interactive interface from which to query various states pertaining to a crash (or breakpoint).

The other perspective is that a crash dump (snapshot) of processor states can be combined with more domain and application-specific knowledge and quickly generates reports and other documents to a wide assortment of specialists and can thus serve as an important glue between actors in a larger development effort or organization. Both of these are relevant to the extent that they cover a broad and detailed description of the states and data of a system or subsystem that, if the underlying cause is proximate in time to the trigger that generated the snapshot or crash, may encompass sufficient data for successful troubleshooting.

In addition, crash dump analysis, as both a manual process and in the form of automated tools, finds its relevance when the target instance is not immediately accessible at the time it presented some anomalous behavior. This further assumes that these snapshots are both generated and collected. This, in turn, implies that a
larger support infrastructure is needed, one that manages all the aspects of collaborating with different stakeholders, and thus ascertains that relevant and intact snapshots are obtained.

Some concluding points in regards to crash dump analyzers:

1. The success of analysis is largely dependent on how intact the underlying data is, meaning that data corruption that specifically affects control structures (metadata) and other key areas rapidly reduces the chances that a snapshot will be useful.

2. The success of analysis is also largely dependent on how encompassing the data is. Some state holders external to the subsystem in question may not be fully retrievable at the point when the snapshot is being generated, such as OS file, socket and pipe descriptors.

3. Crash dump analysis is in many respects similar to computer forensics. Thus, new developments and techniques that benefit the one may well apply to the other.

4. A considerable challenge is to gather only relevant data (both cases) and present only that which is necessary (latter case) for each group of stakeholders, using the most fitting native and non-native representations.

5. The principal problem for the crash-dump analyzer (and to a lesser extent, the tracers) is that of privacy, i.e. how can state relevant for debugging be separated from data that is sensitive to the user.

7.3 Moving Forward

A major development in terms of tracing is the expansion of (virtual-) machine support in the direction of tracing frameworks [51]. Recent works that does this include dtrace [52], systemtap [129], PinOS [53], and Lttng [51]. The basic idea is that the code is prepared with instrumentation points. These can be implemented through, for instance, NOP instructions, which should preferably have been added already during compilation but which may have, of course, been added dynamically. During regular execution, these instructions are executed. By definition, they impose no real change in system or processor state and their only real overhead is the trivial cost of a few more bytes of code. When an instrumentation point is later activated, the instruction is changed to a jump to some stub code controlled by the tracing framework. Note that at this stage, the change is very similar to how software breakpoints in a regular debugger behaves. The key difference lies in what happens after execution has been hijacked.

With the tracing framework, the analyst specifies which tracepoints he or she wants activated, using domain specific language native to the framework. This specification also covers which tracepoint-associated data that should be collected, and how this data should be processed and presented. Thus, among the key differences between the frameworks is the interface specifications for data adjacent to the instrumentation point, and how the gathered information is exported.
A major problem that persists from both tracers and the other tool categories is how the gathered data are represented, particularly when source-code and debug-builds are not available or sufficient.

Even though there are polished user interfaces available, such as Apple instruments [130] and the ltng’s viewer, the actual presentation consists of a fairly simple and static selection of 2D graphs and histograms, similar to those used by many profilers.

### 7.3.1 Trace-probe Networks

The suggested enhancement to tracing frameworks, concerns loosening the grip and focus on the specifics of how each individual trace probe operates, and instead focus on how they are modeled and how collected traces are represented. This means that they do not all need to operate on the same mechanisms but should instead be configurable to a larger degree. If such restrictions are dropped, it would be easier to apply them to a more heterogeneous environment where the set of available and accessible control mechanisms is constantly shifting. This would also make it unnecessary to rely on being able to modify the code of the system or subsystem that we are interested in observing.

![Figure 7.2: Key actions for a trace probe.](image)

We begin by outlining a more abstract probe and breaking it down into a few key functions, as illustrated in Fig.7.2.

**Probe Design**

The first challenge for a tracing probe is interception, which concerns the issue of the extent of control needed, in respect to the extent of control that is necessary, in order to gather measurements. If it is dangerous or otherwise unwise to have a probe alter the space in any way, this function may have to be reduced to a statistical approach and external sampling. When control has somehow been intercepted, the target has irrevocably been altered, which is illustrated by the observer effect. Preexisting debugging interfaces can, of course, still be used to intercept execution, but there are other tools that can also do the job, tools such as the dynamic linker through facilities such as LD_PRELOAD in the GNU LD...
linker (and others). Other viable options are more exotic techniques such as those described in [131], as well as the interface for process or raw memory access, specialized drivers, loadable plugins, JIT facilities of virtual machines [54], process parasites [132], packet filter DIVERT rules etc.

After control has been intercepted, it is possible to sample, meaning to extract data from the subject. Sampling is closely tied to the chosen interception mechanism because it can provide reasonable assumptions as to the type, size and location of adjacent data. As soon as a sample has been generated, the probe emits the sample and forwards control back to the target. The task of emitting a sample involves tagging (time-stamps, sequence numbers, etc.), packaging (compression, encryption) and synchronization.

Note that a key-decision in the design and development of each probe, is to establish which functions that should execute as part of the target, and which should be made external. The external part, is referred to as the control-interface (labeled as control in the figure) and acts as an mediator between the target and the analyst. This part is responsible for all administrative tasks such as attaching and detaching one or more probes (performing the initial interception) to one or several targets.

![Figure 7.3: Coordinating a network of trace probes.](image)

**Probe Coordination**

In order to get such a framework to operate in a more advanced and heterogeneous environment, we need an overarching structure that enables coordination, as illustrated in Fig. 7.3. This can be implemented recursively by establishing hierarchies of controller-interfaces. Similarly, to the workings of each probe, there are a few key functions that cycle, and they can be divided into an outer, external, ring (configure, deploy, activate, teardown) and an inner, local, ring (represent, gather).

The first function, configure, works as an interactive starting point in that the analyst specifies which configuration of sensors that he or she wants. Such a configuration entails the interception mechanism and its parameters, the direct address of the control-interface and the relative address from there to the target. In this way, this step is comparable to the configuration mentioned earlier in this section. At this stage, it is possible to perform both a sanity check (are all the desired targets reachable, can each control-interface inject the probes in question, etc.), and a test-run to make sure that all probes can perform an intercept to detach sequence.
When a workable configuration has been determined, it can be propagated to the control-interfaces in question, i.e. *deploy*.

The inner ring can be activated as soon as samples from trace probes are starting to gather. In the inner ring, there are two key actions that can be alternated interactively. The first one, *gather*, concerns making an informed selection from the data that have been gathered thus far. The second action, *represent*, is necessary in order to make the selected data intelligible by providing visual models to which the data can be attached. These *representations* can be both native (using the symbols and relations from the running configuration) but also non-native with help from a wide range of visualization techniques such as graphs, diagrams, bitmaps and histograms, etc. These representations can also be recorded and distilled into reports akin to the ones previously discussed in the section dealing with crash dump analysis.

The last stage in the outer ring (which subsequently will interrupt the flow of the inner ring) is *tear down*, meaning that all control-interfaces deactivate, disconnect or otherwise disables the probes that they govern, so that the flow of samples is terminated. This action can be followed by a new iteration, i.e. refining the previous configuration based on experiences gained from the previous session.

**Discussion**

The fact of the matter is that similar approaches have already been realized in a number of other contexts; Network communication for many of the communication protocols widely used when it comes to routers and packet filters (firewalls), has dealt with dynamic systemic debugging challenges for a long time, particularly from a performance perspective. In addition to this, there are many refined monitoring tools that make use of similar principles to the solution that was just discussed. Wikipedia has an extensive summary of such tools at [133] and a comprehensive overview of useful native and non-native representations can be found in [86]. Thus, the principal contribution and step forward would be found in unifying such monitoring systems in iteratively refined epistemic models with configurable non-native forms of representation that compensates for the role of, not only the tracer, but the other principal debugging tools as well and that for systemic debugging purposes, at least, there is a serious need to be able to correlate and evaluate causal links between highly different and non-obvious semantic levels and not just at the more convenient and accessible interfaces (e.g. network communication).

### 7.4 Testbed for Systemic Tracing

Moving from the sketch-ideas in Sect. 7.3 to a working prototype not only for demonstrating the feasibility of the concept as such but also to be used as a stepping stone in bootstrapping a robust systemic tracing tool, involves a few challenges of its own. Aside from the issues put forth in Sect. 7.2.1, debugger tools suffer from the requirement of truthfulness more than anything else – we need to know that the output being studied is the actual sampled data of the subject rather than partial or full-corruption induced by the tool itself. *A misleading de-
bugging tool is worse than no tool at all and it is thus reasonable to assume that the software analyst that encounter problems with the debugging tool, is more likely to switch to an alternative one (or change his or her approach altogether) than he or she would be with other types of software.

We took the set of desired initial features and looked for alternative communities with as large an overlap as possible, ultimately settling on graphical frontends for games, emulators and home-theaters in order to quickly reach a reasonably stable core and API, as the feedback loops in these areas are short and it is comparably easier to find workable qualitative and quantitative evaluation scenarios than it would be for a normal debugging tool, hence more useful data from initial testers. Other aspects of this decision will be covered in Sect. 7.4.1.

Thus, there are two tools available: Arcan and SenseEye that shares a majority of the same codebase, with the core of Arcan being used as the baseline, subject to only very modest changes to its current design with a priority of the engine remaining portable (Win32/BSD/POSIX) and scripts being compatible with future versions. The other tool, SenseEye is subsequently a fork of the Arcan codebase exposing a different API, more forms of visual representations, and a more capable frameserver / set of hijack features. Since this will cover somewhat unknown territory (domain-specific languages for dynamic real-time monitoring in contrast to a common scripting interface such as LUA), it will be subject to more radical change as things progress.

A quick rundown of the major components can be found in Fig. 7.4. To map this to the Fig. 7.3 model, the sensor configuration is specified through the scripting interface (configure), and for each specific target (e.g. process, device, dtrace script, ...) a corresponding frameserver is launched, having its privileges dropped to a minimal level (deploy) with named shared memory acting as low-level IPC. With the shared memory mapped, a few basic data-structures and synchronization primitives are set up (ring-buffers, event-queues, semaphores, ...) which are polled on a best-effort or rate-limited basis by the main program and then mapped to resources (represent) exposed to the Scripting Interface, with a lot of options for caching and post-processing (weighted deltas, GPU shaders for GPGPU style offloading, ...). Failure or misbehavior using semaphore synchronization causes the frameserver to be forcibly terminated (teardown).
The *hijacklib* represent a specialized version of the frameserver API in the form of an injectable shared object. The way it is used in Arcan (and in the experimentation further down) is that it intercepts some common symbols related to Audio/Video/Input and either manipulates the data *in situ* before passing it on to the target, or sample and expose to the user script.

### 7.4.1 Experimentation

The sensor explored here will be represented by the *hijack-lib* (Fig. 7.4) used as an example of, if not a worst-case scenario, at least a difficult and data-intensive scenario in terms of observer effect (Fig. 7.2) to set an upper bound for more specialized sensors and to gauge what the capacity of the multi-process frameserver approach would be, and the subject at hand will be the MAME\(^4\) emulator suite emulating the “Top Gunner” game (1986, Exidy). A more detailed view of the specific sensor is shown in Fig. 7.5. Note that every processed video-frame would impose at the very least two additional context switches along with associated inter-process semaphore synchronisation. Also note that there is an element of tampering in this sensor as well since I/O events can be scripted and interleaved with those in the event loop of the target.

![Diagram](image)

**Figure 7.5:** Overview of probe used in experiment.

The reasons for using an emulator as an experiment subject for this kind of benchmarking is due to accurate emulation being notably resource intensive (the emulation of very old devices can still saturate even modern CPU cores and at higher resolutions, the emulator fully consumed one of the available CPU cores) but with little to no dynamic / adaptive behavior or external dependencies present and well-defined strict performance characteristics (set number of frames to deliver at precise intervals). The highly interactive gaming element allows for easy quantitative and qualitative evaluation, e.g. are there any discernible differences in gameplay, and would a comparison of instrumented input/output recordings yield any measurable difference?\(^5\). In contrast, modern gaming engines are highly adaptive to the point that a well-designed engine masks temporary stalls imposed by offsetting image quality which, in addition to ethical cheating/ DRM

\(^4\)http://www.mamedev.org

\(^5\)The reader is encouraged to download and try it for him/herself.
circumvention concerns, make them ill-suited for experimentation. Furthermore, the actual implementation of an interpreting emulator is notably similar to those found in other Virtual Machines (e.g. Dalvik in Android) but with considerably less forgiving constraints.

The reasons for the choice of this particular emulator is its long legacy (in development since 1997), priority on accuracy in favor of speed, well-tested and ported to a large number of different platforms, with source code readily available. The reasons for the specific choice of emulation target is that its distribution and use in this emulator have been sanctioned by the copyright holders and is thus legally available and readily accessible for verification purposes, and the original hardware platform used a vector display which yields a renderpath in the emulator that would be in disfavor of the sensor sampling technique used.

Setup

The script used to configure the tool is the ‘internaltest’ part of the main Arcan distribution. Measurements are gathered based on the `clock_gettime` using `CLOCK_MONOTONIC_RAW` on a 64-bit Linux 3.0 kernel running on a Intel Core2Duo E7500@2.93GHz with frequency scaling and other adaptive hardware tuning disabled. The targeted software itself have not been altered in any way and is thus considered non-cooperative, non-hostile (no antidebugging or similar techniques used). Each run covers 3600 samples (60 seconds runtime) with the timing of each frame stored in a ring-buffer, saved upon termination. The timing for the instrumented run is measured at the point where the corresponding representation has been activated, i.e. the draw-call for the associated video object. The timing for the non-instrumented run is gathered before initiating the readback of the framebuffer from the GPU.

Results

The measurements gathered can be seen in Table 7.1. Note that the data-rate represents data passing between the sensor and the main application and that the actual overhead imposed is at least twice that of the specified rate. The standard deviations are set against the reference clock of (16.67 ms per frame). The instrumented delta refers to the average time elapsed between frames.

<table>
<thead>
<tr>
<th>Res.</th>
<th>Rate (MiB/s)</th>
<th>σ (Non-Instrumented)</th>
<th>σ (Instr)</th>
<th>Mean Delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>320x240</td>
<td>17.578</td>
<td>0.662</td>
<td>1.154</td>
<td>16</td>
</tr>
<tr>
<td>640x480</td>
<td>70.313</td>
<td>0.576</td>
<td>58.78</td>
<td>20</td>
</tr>
<tr>
<td>800x600</td>
<td>109.863</td>
<td>5.523</td>
<td>226</td>
<td>30</td>
</tr>
<tr>
<td>1024x768</td>
<td>180.0</td>
<td>3.249</td>
<td>2457</td>
<td>65</td>
</tr>
</tbody>
</table>

Table 7.1: Measuring the observer-effect on an instrumented subject (in ms).
Discussion

As evident from Table 7.1, the standard deviation from the ideal mean differs quite notably between the instrumented run and the non-instrumented ones, especially as the resolution increase. This is because the sensor was not equipped with any other synchronization options (e.g. dropping or batching frames to account for temporary stalls, pipeline bubbles etc.) thus as soon as one frame starts drifting, it cascades onwards. Using the mean delta between frames, we can see that already at 640x480, every 4th frame would’ve have to been discarded to maintain transparent sampling of target state without a user noticing. If one can accept asynchronous requests through PBO (Pixel Buffer Object) transfers, these stalls can be circumvented and effectively allow much higher resolutions and data rates.

To translate this sensor into a more serious context, one can think of the sampled framebuffer (emulated video display) as any memory region, heap-allocated or otherwise memory-mapped, and by altering offsets between sample points to, for instance, match minimum page-size combined with abusing certain syscalls (e.g. `write`) quite rapidly map out process memory space layout and access patterns while using GPU shaders with a lookup-table in the host application to efficiently track changes and alternate between non-native representations for real-time analysis. Other fairly trivial use cases for a similar sensor would be to project the sampled buffer into a point-cloud and visually assess distribution for cryptography engineering purposes – most, if not all, necessary code for these suggestions are already in place, with the major point being that this can be done from several processes in parallel, overlay with streaming data from other analysts working in parallel, while the session is being recorded for off-line tracking.

7.5 Concluding Remarks

The systemic shift illustrated in Sec. 7.2 is not binary in the sense that more traditional debugging (whatever that is) somehow becomes extinct. Instead, the scope of the overarching task is widened. With this follows that the challenge of analyzing an anomaly also demands, not only the perspective of the feature-phone (that is still relevant in the kernel-space) but also a profound knowledge in reverse engineering, de-obfuscation and similar techniques, along with deep rooted understanding of the interplay between debugging and security and DRM related protections, an understanding that also includes cryptography engineering.

Furthermore, it is worth noting that not only is it important to have intimate knowledge about the mechanisms and limitations of the individual tools, it is equally essential to be able to coordinate these into a larger chain that covers the entire spectrum. Late stages of debugging is still likely to be needed for a long time, so it is a good idea to prepare for this in advance by abstaining from solutions that counteract the aforementioned tools and methods (prebugging).

In this chapter, we have argued that many modern development and protection techniques counteract mechanisms that are needed for crucial debugging tools to be useful in their current state, which further complicates the already difficult task of fixing complex problems in increasingly heterogeneous and dynamic software
solutions where multiple stakeholders fight for control over enabling platforms. We have also highlighted the importance of evaluating these tools in the context of the wider organization that will directly or indirectly be forced to operate them. We thereafter presented a taxonomy of problems facing specific debugging tools derived from seminar sessions with domain experts and, using current tracing frameworks as a model, described enhanced tools for enabling systemic tracing which would enable dynamic monitoring of software-intensive systems for a wide range of purposes, the more interesting of which would be complementing and validating protection mechanisms for existing static hierarchical monitoring systems (e.g. SCADA for critical infrastructure).
8 The Proper Role of Agents in Future Resilient Smart-grids

8.1 Background

The management, distribution and transmission of electricity as provided by the power grids of today is safeguarded and managed by a wide assortment of proprietary, closed and hierarchical control systems with the central source of intelligence and authority by operators acting upon information from legacy-rich static monitoring and command systems, i.e. SCADA. Even though this form of operation has arguably served us well thus far, we are closing in on the tipping point towards more open and dynamic solutions, i.e. those that fall within the ‘Smart grid’ Smart-Grid (SG) umbrella. Among the principal new requirements on the future grid that would demand such a transition, we find:

- A need to enable both horizontal and vertical integration of generation and distribution of power.
- Emergence of new and flexible sets of stakeholders requiring horizontal integration of shared situation awareness and control.
- Integration of information systems related to energy flows, business processes, as well as smart configuration and monitoring of related systems.

The principal benefit that can be gained from fulfilling these requirements is the utilization of vast amounts of Distributed and Renewable Energy Sources (DER/RES).

Arguably, implementation of needed distributed intelligence supporting flexibilities in connectivity as well as loosely coupled coordination patterns enabling detection and sustainable response, requires a careful selection and adaptation of agent technologies and configurable infrastructures. Resilience and Interoperability are two important systemic properties of Smart Grids that have to be assessed and clarified. For instance, self-healing could be addressed and solved in a proper way provided that a stated invariance relation based on resilience and interoperability properties is maintained. Setting up and validating such invariance is dependent on a suitable experimental platform, such as the one we suggest in Sect. 8.7.

In this chapter, we first introduce the conditions, restrictions (Sect. 8.2, Setting the Smart-Grid Scene) and threats (Sect. 8.3, Antagonistic Threats) that are likely to
hinder the successful introduction of agent empowered distributed intelligence in future Smart-grids (Sect. 8.5, Resilient Smart Grids). This setting, Sect. 8.3 – Sect. 8.6, is then used as input to the design of an experiment scenario supporting setting up and validating agent empowered self-healing of a microgrids, Sect. 8.7.

### 8.2 Setting the Smart-Grid Scene

The power grid is an interesting critical infrastructure since its main output, electricity, is crucial in the operations of most other critical infrastructures and services that modern societies depend on. The complexities involved in operating and maintaining these infrastructures, however, depend on the trustworthy presence of a large assortment of information and communication technologies (ICTs), which in turn relies on electricity from the grid, forming a circular dependency [55].

Other complicating circumstances for operating the grid as an infrastructure include:

- Massively distributed.
- Brittle, with a unidirectional structure for generation, distribution and transmission with many "single points of failure".
- Lacks a safe-state, i.e., cannot be reverted to a previously known state after a systemic failure (e.g., a blackout).
- Numerous stakeholders with dynamic, and at times conflicting, goals and behaviors, acting in international markets where available supply is bought, sold and traded between actors according to different regulations and constraints.

Thus, solutions that cater to harness most or all of these caveats are paramount to the success of Smart-grid initiatives. In short identifying resilience as a key property for future Smart-grid deployment and maintenance. Resilience, in our setting, refers to systems that have ductile and self-healing properties while maintaining well-formed invariants of Key Performance Indicators (KPIs). The systems of systems view of Smart grids entail compositions and coordination of smart software and hardware components in a service-oriented system view.

In this chapter, we argue that models and mechanisms exploring increased software and communication resilience is a key enabler towards distributed intelligence and therefore towards future Smart grids. Addressing Interoperability, issues are guided by the GridWise Architecture Council Interoperability Framework [134].

More precisely we focus on smart monitoring enabling different forms of self-healing. To that end we will investigate state-based models and information modeling and controlled message exchange between components, with emphasis on the proper use of multi-agent technologies within such a setting (Sect. 8.5).
8.2. Setting the Smart-Grid Scene

8.2.1 Monitoring Smart Microgrids

A specific vision of one kind of smart grid is the microgrid. In microgrids, the topology is a mesh of operationally independent, but interconnected cells. In the INTEGRAL project [105], the following three different modes of operations for a cell, supporting aggregations of massive amounts of DER / RES, were defined as follows:

1. **Normal** operating conditions, showing the potential to reduce grid power imbalances, optimize local power and energy management, minimize cost etc.

2. **Critical** operating conditions, showing resilience opportunities also in integrated grids. Resilience could be modeled as detection of a critical state followed by a self-healing action to bring the system back to a normal state.

3. **Emergency** operating conditions showing self-healing capabilities.

The practical details of implementing support for restructuring current power grids into smart grids relies heavy on expanding the underlying metering structure, hence the steady introduction of Smart Metering technologies worldwide. The extent of capabilities and regulating policies regarding deployed smart meters vary between countries, and suggestions for their final role range between being additional data-points for increased precision in demand forecasting, to acting as local smart control-systems.

In order to identify and monitor the **Normal**, **Critical**, and **Emergency** states with corresponding system hardening to achieve increased resilience, we address Antagonistic Threats in Sect. 8.3.

In Sect. 8.5, the role of Smart metering systems is covered both as a tool supporting customer empowerment and as a new kind of vulnerability of Smart grids.

In Sect. 8.6, our analysis of vulnerabilities is summarized in two principles towards ensuring increased resilience of Smart grids:

- Proper data models of components and their interaction in Smart-grids are crucial. Candidates include the standards IEC 61850 and IEC 61499.

- Proper Architecture models, for instance ANSI/ISA-99, supporting network segmentation into security zones (DMZ) with information exchange across specified conduits.

These principles supports defense in depth based strategies based on context dependent intelligent multi agent systems covered in Sect. 8.6.1, Resilient Information Processing Systems. Assessment and validation of proposed solutions for critical situations are presented in Configurable Experiment Platform, Sect. 8.7. The configurable test-bed is supporting fault injection and monitoring at selected interfaces is described in Sect. 8.7.1. Our Results and Concluding Remarks are given in Sect. 8.8.
8.3 Antagonistic Threats

Before the advent of cyber security as a central concern, threat and risk management models for the power grid were primarily focused around environmental (weather, component wear), design (deployed structure and restructuring, load balancing, generation, etc.) and challenges inherent to power systems (transients, short circuits, etc.) with related issues in the ICT taking a more distant role.

With such a perspective, antagonistic threats (break-ins and physical sabotage, theft of property, etc.) were dealt with through physical barrier protections having a corresponding separation within the ICT. It is, however, well-known that this model has failed [127]. Thus, a precondition to all other advanced forms of SG deployment is that future ICT should account for both aforementioned threats to previous grid structures but also encompass more sophisticated antagonistic threats.

Sabotage

With critical infrastructure as with any high-end industrial endeavor, there is a considerable assortment of political, economic and ideological interests in play that may come to target the Smart Grid specifically. The Stuxnet attack [101] provides an interesting perspective on sabotage through manipulating SCADA systems, showing that ICT can give benefits to an attacker that more military intervention cannot.

Some key properties of the Stuxnet attack are thus:

- **Precision Strike** – Previous SCADA related incidents can be attributed to rogue operators and poor handling of more generic corporate IT attacks that “by accident” breached the barrier between corporate IT and control systems. By contrast, the Stuxnet payload did little damage outside of the targeted environment, and gradually got more precise (some parts of the attack were missing from the first push), with an embedded expiration date.

- **Multiple Coordinated Levels** – A major part in the complexity of its design is the fact that it takes advantage of a coordinated front. First through social engineering by planting specially prepared storage devices around the target location, relying on a careless (or malicious) operator finding and connecting them to computers within the Demilitarized Zone (DMZ). These devices contained generic (operating system- and common network services), domain-specific (industrial control systems) and target-specific (programmable logic controllers) payloads.

- **Dynamical Updates** – If an external network connection was present, the software could receive updates and export gathered intelligence, making it possible to adapt to after-the-fact protections (anti-viral signature updates etc.) and to iteratively increase the precision of local attacks.
8.3. Antagonistic Threats

- **Certificate Theft** – The software installed after local privilege escalation had occurred was signed using private keys from two major suppliers of core components e.g. network interface cards. If the chain of trust was damaged from the previous stages, it was completely broken at this point.

What distinguishes Stuxnet is thus the coordination of multiple attacks, both on-line and offline, to precisely strike at one well-defined target without directly exposing the aggressor.

**Business Intelligence**

The shadier side of business intelligence in the sense of data harvesting botnets and other forms of computerized espionage is a notable threat for generic corporate ICT, SCADA and SGs. Unfortunately, evidence of such events is scarce at best. Recently however, a few major incidents have been uncovered that shed a little light on the issue, e.g. Night Dragon [102] and Shady Rat [135].

The underlying attack pattern involves breaching some sort of outer perimeter exposed to the general public or specific stakeholders, in these cases through spear phishing (specially crafted social attacks designed to trick the target into running a backdoor program, visiting websites or unknowingly exposing access credentials such as passwords) but also from common web-service vulnerabilities. After breaching the first perimeter, routers, printers, e-mail servers and other central points of data access become obvious targets for gather more intelligence on specific operations and key individuals. Gathered intelligence can then subtly and surreptitiously be leaked to the aggressor through services and protocols known to be used as part of day to day operations at the specific site, subverting anomaly detection.

Even though Night Dragon and Stuxnet covers different objectives and work at different levels of technical refinement, they are similar in the sense that the respective attack progression is heterogeneous and iterative; following a broad and generic initial method of attack, they become increasingly domain specific and targeted. While it is hard to judge the level of active involvement on behalf of the aggressor after deployment of the initial backdoor, the components of Stuxnet required little intervention, while-as Night Dragon represents a more active participation, i.e. Sabotage- kind of attacks can be carried out offline, while-as the Business Intelligence need some level of external communication to expose gathered data.

**Business Sabotage**

The last case, concerns the combination of the two aforementioned threats. Gaming consoles and more recently, smartphones are attractive target for both legitimate and illegitimate forms of tampering and represent advanced computers with heavy restrictions on the level of access that end-users have. To enforce the restrictions placed, a wide range of costly protection mechanisms are added, ranging from encrypted and signed code to custom storage devices and tamper protection.
There is, however, a conflict of interest between the manufacturer of the device controlling which code is allowed to execute and, the end-user that wishes to run software written by themselves or other enthusiasts. In order to do so, the aforementioned protections have to be circumvented somehow (often referred to as jail-breaking) and the general process is comparable to the privilege escalation performed as part of the larger attack in the Stuxnet case.

Among current-generation consoles, the security system of the Sony PlayStation 3 had until August of 2010 not been compromised in any meaningful way when modified USB host adapters appeared (e.g. PSJailbreak). These devices automatically performed an advanced attack on the device during start up, allowing signed code from untrusted sources to be executed, allowing for piracy. This was followed by [136] where a severe flaw in the underlying cryptography engineering was uncovered, making it possible to deduce keys usable for signing code, irrevocably breaking the chain of trust.

The exact details, causation and time-line of the aftermath is not fully known, but what has been publicized is that the attempts at remedying the situation stirred an uprising among internet activist groups leading to attacks that publicly exposed millions of customer records and a denial of service of the network with downtime ranging from a few weeks to several months depending on geographical region, resulting in considerable loss of revenues.

The business sabotage case is interesting in a smart-grid context when used as an analog for smart meters in the sense of trusted devices acting as part of the larger monitoring system, placed in an untrusted and possibly hostile environment. Although the processors and other circuitry of gaming consoles are several magnitudes more powerful than current generations of smart meters, the console hacking activities well illustrate the kind of tampering that smart meters need to withstand, at a fraction of the cost per unit.

### 8.4 The Smart Grid Angle

When the presented cases are combined into a broader picture, the difficulty in protecting the SCADA perimeter from both external and internal threats is clear. While the role of the Smart Meters have been scrutinized in other contexts [56], the threat may well turn out to be both horizontal (between prosumers) and vertical (prosumer / utilities), as shown with the PS3/Business Sabotage case (cheating between users, piracy between users and distributors); similar (although slightly less embarrassing) cryptography implementation flaws have already been found in parts used by at least one brand of Smart Meters.

Combining economic incentive stemming from new business models in the Smart Grid (making profit from manipulating consumption and production), the homogeneity of the Smart Meters (simplifying the process of making wide-spread worms, etc.) and the likely involvement of essentially small-scale SCADA systems for managing devices in homes (Smart Houses) and the Smart Grid looks to be a hotbed of security concerns well worth further study before even more wide-scale use takes place.
8.4. The Smart Grid Angle

8.4.1 Rogue SCADA

Working from a key characteristic of SCADA, i.e. its data driven nature, the first theme to cover thus concerns possible threats against firstly, the validity of said data and secondly, the possible use of the data – but these two principal threats overlap.

The generic idea is the controlled compromise of a singular node or link from which an attacker can alter, introduce, delay or remove information destined to another node or link within the SCADA.

To specialize this description into three separate levels:

- **High-level**, While internally, a SCADA system resembles a tree- (directed, acyclic graph) like data structure with the MTU as a root node in its simplest of forms, but more advanced structures where they nest (SCADA within SCADA) or interlink (exchanging data) are also possible. Among the likely candidates on how a SCADA for the smart micro grid have to operate would include a SCADA for each local cell, but also one or more at an aggregated level. With no other option to ascertain the validity of exchanged data, the linked SCADA systems are implicitly trusted and a rogue cell or sub-SCADA can export data that has been manipulated to favor or disfavor a specific cell.

- **Mid-level**, Data that pass between SCADA and other parts of the organization or company may serve as the foundation for key business decisions, bidirectionally; Information exported from a SCADA containing operational parameters (disturbances, capacity, loads...) while information imported would assist the decision making of the operators. Rogue translation between these two data domains can disturb operation-sensitive decisions at two different ends. Furthermore, in line with the business intelligence situation, exported data can be copied and moved through external channels.

- **Low-level**, Concern the core SCADA components and their respective communication in two different ways. The first way represents rogue nodes presenting manipulated sensor readings at any level in the internal chain of trust. These readings can be outright wrong or slightly manipulated in some more subtle way, e.g. least-significant bit-flips but updated checksums), replayed, shifted time-stamps etc. The other way is on the command-level, faking responses that a command was carried out correctly even if it had been discarded or ignored.

In its most refined sense, a capable attacker controlling all these levels could essentially construct an abstract form of SCADA of his or her own, using only the bits and pieces of existing ones. Mode 2 and 3 of the experiment environment in Chap. 6 corresponds to the Mid- and Low-level settings by a compromise in the communication infrastructure. The experiment in Chap. 7.4.1 corresponds to a Low-level setting applied at the HMI layer.
8.4.2 The Empowered User

While as the Rogue SCADA setting was explicitly directed towards the many data processing stages which when combined essentially defines SCADA, and ways in which such an attack can be carried out are easily envisioned (burglary, compromised supply-chain, rogue employees, software vulnerabilities etc.) their effect could also be modeled as a software bug among many others (e.g. the linked list issue of the NE blackout of 2003 [103]). The empowered user scenario on the other hand, is related to the swarm-like problem of botnets, i.e. large numbers of compromised nodes acting in unison against a specific target, with many such nodes being well out of the control of any specific vendor.

This is particularly interesting when viewed from the smart grid and smart metering angle – lack of education, ability or sheer ignorance have allowed botnets of millions of compromised nodes to be constructed, many through simple social engineering techniques in the form of a payload attached to an e-mail with a message encouraging a user to run it. As with the business sabotage case presented, quite sophisticated exploits can be molded into easy to use packages requiring little to no expertise in performing.

The reason such a swarm can be constructed stem in part from the homogeneity of its individual components, thus if a reliably exploitable vulnerability can be found in one component, foreign code can be introduced in all that share the characteristics of the vulnerability (Virtualization for compatibility as per Chap. 5). Subsequently, there are several dynamic protection mechanisms in use (e.g. Address-Space Layout Randomization (ASLR)) that reduce the likelihood of exploitation of buffer overflows, but consider, for instance, a mandated software update for metering components signed using a stolen certificate.

As hinted in Sect. 8.4, likely developments in Smart Houses, with the home owner being put in charge of a real-time monitoring system over electricity, water and gas consumption, with the ability to turn of and on devices and change smart meter policies remotely, from the comfort of his or her own laptop, tablet or smartphone, will open up a wide yet interesting angle on the SCADA security challenge.

8.5 Resilient Smart Grids

When the presented cases of Sect. 8.3 are combined into a broader picture, the scope of the challenge in defining and protecting the smart grid perimeter from both external and internal threats is clear. While the role of the Smart Meters have been scrutinized in other contexts [56], the threat may well turn out to be both horizontal (between prosumers) and vertical (between prosumers and utility companies), as shown in Sect. 8.3 (cheating between users, piracy between users and distributors) and similar cryptography engineering flaws have already been exposed in key components used by at least one brand of Smart Meters.

Even though such attacks may in their initial form require a high level of technical sophistication and skill to develop and deploy, they can later on be automated and packaged in such a way that a comparably unskilled end-user can replicate
the same attack – the number of people using alternative markets with jailbroken smartphones alone are in the order of millions. The surrounding communication technologies are, furthermore, subject to a number of practical and publicly demonstrated attacks, including those targeting other infrastructure e.g. cellular networks using GPRS/GSM, coincidentally one path in-use for conveying smart metering data and software updates for metering devices.

If data from these meters are being used as sensory input for agents trading on dynamic markets or acting as decision support for operators attempting to fix a cell in an emergency state, they are very likely targets for attacks similar to the ones described in Sect. 8.3, but also, due to the economic incentive, as an aid for manipulating markets as per Sect. 8.3. Our suggestion to harness those threats is to aim at defense-in-depth solutions based on the identified principles of Sect. 8.2.1 as outlined below.

8.6 Moving Forward

In the previous sections, the focus has been on illustrating the depth and breadth of current and coming problems of maintaining and securing future smart grids: the mere inclusion of quite standard ‘after the fact’ pattern-based approaches will be woefully inadequate against the kind of adversaries now known to be present.

The advantages in defending Smart-grids however, is that there are systemic properties that assist in leveling the playing field; because of the data-driven nature, the most well-known entity is in fact the data-models on which all components involved indirectly operate.

Even though a great deal of effort have been placed on making sure that sampled data can be retrieved within a set threshold and arrives intact, this is rather within the domains of any modern communication technology. If, however, the actual data-model (e.g. IEC61850/IEC61499 standards) of the governed structure is verified, locked down and integrated into both the processing components and security-centric ones stronger benefits can be reaped alongside possibilities for easier external validation of processing components.

Considering the sheer number of cooperating and competing technologies both old and new, combined with the inherent complexity of the overall monitoring task and the nature of the presented threats, there is strong pressure placed on the composition and accuracy of testing- and experimentation-facilities, particularly to study the interactions between security mechanisms, monitoring systems, maintenance operations and the governed infrastructure.

For instance, there are suggestions that involve encrypting the communication between all participating devices, but may ultimately prove infeasible with current hardware and latency restrictions, and would furthermore require a key management structure and subsequent training (revocation procedures, etc.) and updating IDS and other protective devices monitoring to be able to decrypt such communication (else they would be rendered useless).

Other suggestions include building the final solution using web services (reducing the usefulness of traditional firewalls, increasing the need for more costly
higher-level equivalent filtering) with data expressed in markup-languages that are difficult to build robust real-time parsers and processing around (possible infinite buffering, defining filtering heuristics for a large set of possible text-encodings, etc.).

In short, the larger design challenge may not be the mere development of strong protective mechanisms but in fitting these together into a comprehensive architecture (e.g. ANSI/ISA99) such that they neither hinder normal operations nor cause incompatibilities, yet still provide necessary protection.

8.6.1 Resilient Information Processing Systems

Given the history, scope and complexity of traditional grid structures, it is unlikely that the transition towards smart micro-grids would happen overnight, but rather as a gradual shift over an extended period of time where key information processing components will be adapted and reconfigured, rather than rebuilt. This implies that there are no accurate, discreet and formalized requirements that would capture the end system or its intermediate steps, but rather that verification and validation efforts have to work from the perspective of an insufficiently specified system.

Thus, we face the risk of introducing non-linear phenomena due to unspecified combinations (resonance) of interactions. Given the brittle state of present power grids (Sect. 8.2), resilience engineering and corresponding mechanisms need to be introduced in order to resolve this barrier. Among such mechanisms that can be used to make an information processing system more resilient, we find self-healing and monitoring [1].

Smart Self-healing requires smart monitoring and smart response! That is, basically two types of support tools; firstly empowerment of operators, secondly smart control and detection of information exchanges across interfaces. We argue that those goals can be approached by identifying the proper roles of agents in future resilient Smart Grids.

8.7 Configurable Experiment Platforms

Based on the requirements, problem specification and threats from Sect. 8.2 and Sect. 8.3, we have, within the INTEGRAL- project, developed a smart/micro-grid testbed together with G2ELab/IDEA in Grenoble, France [11].

Fig. 8.1, depicts an overhead view of the major components involved, which comprise an ICT/monitoring system operating on a scaled down microgrid cell complete with loads, transformers, distribution network, DER/RES and SCADA.

The major components of the SCADA system are RTUs (usually located in transformer stations), responsible for opening or closing breakers, and sampling sensors. These RTUs are connected to a Main Terminal Unit with database storage (Historian) and an operator interface (Human Machine Interface, HMI). For a more detailed view of the underlying engineering effort, please refer to Chapter 6.
To configure a testbed with self-healing capabilities as in [57, 58], there is an Agent, a network of FPIs and a FR. The FPIs are monitoring devices that can approximate the location of a fault. The Fault Recorder subsequently logs sensor data. The Agent scans SCADA topology, interprets FPI data from FR in order to calculate and suggest a remedial intervention to an operator.

Controller, Control Nodes, VPN and Security Lab are part of the ICT experimentation setup, and its primary role is partitioning the communication network into controlled, separated, networks – allowing for fault injection and experimentation with communication characteristics (traffic shaping).

### 8.7.1 Testbed

The ICT experimentation setup has the following modes of operation:

1. **Local Operation** when VPN uplink is disabled and control nodes are only used for their monitoring and traffic shaping abilities.

2. **Remote Monitoring** when all traffic is transparently mirrored to the security lab through an encrypted tunnel.

3. **Remote Tampering** when all traffic is redirected to the Security Lab, manipulated or corrupted then returned.

The Security Lab connection, in Sweden, can be configured to represent antagonistic threats or act as interface for additional microgrid cells, allowing the testbed to span several cells. Thus, by alternating between these modes it is possible to play out a number of scenarios based on the aforementioned threats (Sect. 8.3).
Unfortunately, since the SCADA also exhibits the same properties and vulnerabilities of real-world SCADA systems, in the sense that it is designed to operate in a secure DMZ with little to no safe-guards in place, it limits the number of actual interventions that can be tested until sufficient verification and validation efforts have taken place.

8.7.2 Experiments

Among the principal goals for this testbed is to enable validation and verification of agent-empowered self-healing, to assist the development of dynamic monitoring and protection mechanisms that harness both environmental and antagonistic threats, and to demonstrate the feasibility of self-healing and monitoring as resilience mechanisms in smart microgrid cells facing critical or emergency situations.

Figure 8.2: Latency traces for several iterations of a self-healing sequence.

Among supposed key benefits of introducing agents as complement to operator action alone, would be a decrease in response-time regarding some disturbance, reducing it from roughly a few minutes down to a few seconds. A prerequisite is thus the ability to query a large number of sensors (here represented by the FPIs with the FR acting as their interface), making the reliance on the communication infrastructure a key concern [59].

The three primary areas of interest for first rounds of experimentation were:

- Latency – how large an effect introduced delays will have on the outcome of a self-healing cycle, e.g. if it is possible for an adversary to time an attack in such a way that any self-healing activities are undermined. The ability of the testbed to carry out such manipulation is shown in Fig. 8.2.

- Data / Traffic – in terms of both traffic volumes and actual traffic contents. Given the central role of the underlying data models, both in designing more precise monitoring systems, and for tuning protection mechanisms,
8.7. Configurable Experiment Platforms

it is interesting to see how well the current testbed reflects aforementioned standards (e.g. IEC 61850) and how resistant these are to manipulation from both an informed and from a blind one.

- Information Leakage – Many have raised concerns that the increased precision of Smart Meters can be used to discern otherwise private and sensitive information belonging to individual customers. In this context, we are interested in the reverse, i.e. if unprivileged access can be used to discern sensitive information about the state of the microgrid.

8.7.3 Scenario

To cover as much of the goals of experimentation in Sect. 8.7.2 as possible, the following scenario was devised:

ICT and Microgrid is reset to a stable state, real-time monitoring and data recording is activated. Communication parameters are set (latency, packet-loss and bandwidth limitations between subnets) along with the mode of operation (as per Sect. 8.7.1).

1. Agent connects to the SCADA, retrieves and analyzes topology.
2. An operator injects a fault (three-phase fault) into the grid.
3. The fault is detected by the FPIs/FR and subsequently, by the Agent.
4. The scenario ends when the agent suggests remedial actions.

8.7.4 Results

In regards to latency, the reference scenario, without any active traffic shaping, completed in 6 seconds (steps 3...5). Activating mode 3, corresponding to an attacker redirecting all traffic to a foreign location (stable at a round-trip of 50ms between the sites over the course of a month), yielded no observable qualitative difference in normal operation of the SCADA, yet increased the completion time for the scenario to 24 seconds. It took extreme network conditions (600ms latency, 40% packet loss) to push the completion time to the approximate upper unassisted operator intervention threshold (2 minutes). Concerning captured data, the majority of traffic was, unsurprisingly domain specific to the activities at hand, at rates never surpassing 160KiB/s. However, the available dissectors at the time (e.g. Wireshark) failed to properly decode the majority of the captured data.

In regards to information leakage, Fig. 8.3 shows an annotated graph of a completed full cycle during extreme network conditions from the traffic between controllers in mode 3, but other interfacing points exhibited nearly identical patterns. Even lacking detailed domain expertise, the different stages are clearly discernible from these measurements, showing when, for instance, a denial of service attack would be the most effective. Common counter-measures e.g. padding traffic with random noise would be ill-suited in this situation due to latency and bandwidth requirements.
Figure 8.3: Annotated latency traces from several completed cycles showing use of Mode 3 (remote tampering) to establish a baseline, and testing different degrees of these settings in Mode 1 (local operation).

8.8 Concluding Remarks

We have herein explored the proper use of agents in resilient smart-grids in the sense of:

1. A decision support mechanism complementing, not replacing, mandatory and regulated operator interventions during critical and emergency states.

2. An experiment aid for generating interaction patterns as input for generating data when facing imprecisely specified information processing systems with a strong legacy component.

We furthermore suggested a classification of three principal kinds of threats, i.e. sabotage, business intelligence and business sabotage illustrating each threat with recent real-world incidents. These threats represent a minimal baseline of what any intelligent smart-grid solution will need to account for.

Lastly, we described a distributed testbed allowing experts from different domains to collaborate in experimental validation of the aforementioned use of agents.

Initial results highlighted exploitable vulnerabilities in the current design, and future work thus includes addressing these vulnerabilities and introducing refined real-time monitoring in the sense of systemic tracing facilities.
Part III

Experiences
9 Results

The purpose of this chapter is to give perspective on the contributions from Part II in relation to the structure put forth in Part I and is subdivided into five sections. The first section, Executive Summary gives a high-level summary of the path from problem description to approach, results and impact and is intended for a broader audience. In the other four sections, Overview, Harnessing SCADA, ICT for Smarter Grids and Framework for Systemic Debugging, the hypotheses and research questions from Chap. 3 are revisited and combined with key contributions from the chapters that comprise Part II of the thesis, but also extended with additional results and lessons learned.

9.1 Executive Summary

The main premise of this thesis is the proper use of computing mechanisms in the design and implementation of monitoring systems as part of the active governance of current and future infrastructures. Properties of these systems and infrastructures are similar to that of Ultra-Large Scale System (ULSS) [60].

- **Massive**, no single stakeholder possess a complete view of the components, policies, regulations, protocols etc. involved.

- **Critical**, societies and other infrastructures would suffer a notable or serious loss in the event of a disruption.

- **Organic Growth**, reflecting that of the societies which depend on it and dictated by a balance of political, technical and social considerations.

- **Heterogeneous**, comprised of a wide assortment of components with a long legacy span across generations of technologies, often lacking access to accurate descriptions (e.g. source-code, build-systems etc.) for reliable reproduction or replacement in the event of component failure.

Among the numerous subareas of interest, we focused on:

- **Resilience**, restructuring current brittle infrastructures where a single failure might cascade and bring the infrastructure down as a whole, into resilient infrastructures where disturbances are expected to some degree, and preventative measures have been implemented to harness such disturbances.
9. **Results**

- **Securing SCADA**, hardening the hierarchical control and data acquisition systems used as part of the monitoring solution, towards being able to tackle the security challenges of current and future antagonistic threats [99] [61].

These areas were approached from, primarily, a computing perspective, emphasizing the interplay between virtualization techniques, experiment environments, software security and late-stage debugging. By taking an exploratory and experiment driven research approach we attempted to try and paint a broader picture of the challenges involved while suggesting possible remedial actions.

The targeted critical infrastructure at hand was that of the **power grid**, due in part to its role as a key enabler and central dependency for other critical infrastructures. More specifically, we looked at **smart-grids**, which, briefly put, is an active inter-disciplinary field of research that covers the controlled introduction of advanced computing-centric information processing in order to better respond to fluctuations in market and environment- conditions.

Parts of this work has been done in close collaboration with Grenoble Institute of Technology (G2ELab) through the two European research projects INTEGRAL [105] and SEESGEN-ICT [106], providing a much appreciated electrical engineering perspective, but also with Sony Mobile Communications through a specialized debugging task-force, MiB, providing industrial validation in regards to modern-day challenges in embedded system development, similar to what can be expected in future smart meter\(^1\) ecosystems.

The contributions as presented in Table. 3.1, when combined, argue in favor of the role of **systemic debugging** (Chap. 5, [70]) as a methodological framework and organizational resilience mechanism (Chap. 4) in the cross-section between software-, systems-, electrical- and security- engineering implemented by specialized task-forces that, assisted by **systemic tracing**- and **intelligent agent**-(Chap. 7, Chap. 8) empowered **experiment environments** (Chap. 6), pro-actively design and evaluate **informed system protection** ( [80]) measures and **scenario-driven training** (Chap. 8) as a final means of ensuring Service-Level Agreement (SLA) conformance([87]) in the gradual shift towards smarter grids.

**9.2 Overview**

In this section we revisit the hypotheses from Chap. 3 and show how they map to the remaining sections in this chapter.

1. **It is possible to restructure a preexisting static hierarchical monitoring system through principled virtualization and gain the ability to iteratively perform increasingly sensitive experiments on otherwise brittle subjects.**

This is addressed in Sect. 9.3 by the technically motivated principles of Chap. 5, the application domain from Chap. 4 and the demonstrated use of the environment from Chap. 6.

\(^1\)enhancements to current metering technologies as a key component and enabler of future smart-grids
9.3 Restructuring SCADA

II. To counteract vulnerabilities imposed by dynamic reconfiguration through self-healing activities, existing monitoring facilities should be re-purposed to allow interleaving of different expert domains into combined monitoring models, in contrast to planar ones where different sets of stake-holders monitor only variables of immediate interest.

This is addressed in Sect. 9.4 through the self-healing scenario when combined with interception and traffic manipulation as demonstrated in Chap. 8, with the ICT tampering not being directly observable (other than in hindsight) from the HMI view while the SCADA state not being observable from the provided ICT monitoring.

III. Dynamic, interleaved, monitoring models as suggested by (II), can be constructed through dynamic instrumentation of pre-existing software components without compromising response times in the monitored infrastructure.

This is addressed in Sect. 9.5 by the experiments on virtual machines instrumentation through the development and subsequent use of the systemic tracing prototype, as described in Chap. 7, taking advantage of some facilities of the dynamic linker to intercept a subset of key functions in a third party program to interactively alter its behavior and present alternate data representations.

9.3 Restructuring SCADA

The interest in restructuring SCADA stem from its software-intensive system-like properties (2.3.1), its role in delivering monitoring – key in infrastructure maintenance and upkeep, but equally key as a primary data-source for organizational levels of decision making and rising security concerns.

Part of the security concerns (8.3), stem from the unsustainable segmentation of protection domains or, more to the point, the notion that the control network can- and should- be physically separated from other parts of the corporate network, and from relying on such a separation. With smart-meter deployment shifting the threat model further, the incentive to restructure SCADA has never been stronger.

Given the aforementioned brittle and legacy- rich nature it is, of course, paramount that this is done without disturbing day to day operations (maintaining the online value of the system at hand), while ensuring that any changes don’t expose vulnerabilities or break current functionality, hence the prerequisite need for resilience but also the need for the ability to decouple software from legacy devices.

The research questions connected to this issue are thus:

- RQ1, Which principal mechanisms exist for enabling and improving resilience in software-intensive systems?
- RQ2, To which extent can the drawbacks or caveats associated with virtualization be controlled?

We established virtualization as the overarching means for enabling resilience in software-intensive systems, (Chapter 5). A virtualization is here defined as the embodiment of a subset of computing abstractions targeting one or several of these three key resource groups: storage, communication and computation.
9. RESULTS

A virtualization splits the state space of its machine into two parts, a virtual space and a machine space (environment). The primary activity for a virtualization is thus to dynamically translate between the code of the virtual space to that of the machine space. This phenomenon is recursive as the machine space of a particular virtualization may also comprise the virtual space of some other virtualization.

A key distinction in this regard is that a virtualization is a runtime phenomenon and can thus be both observed, instrumented and otherwise tampered with, and is, relatively speaking, a safer place for instrumentation than in the virtual space (code to be executed) as that code is built to the assumptions of computing capabilities and interfaces exposed by the virtualization rather than that of the underlying machine, if the virtualization ideal (Chap. 5.2) can be sustained that is.

![Figure 9.1: External state-holders and their influence on virtualized resource(s).](image)

We also established primary caveats to control for, based on whichever benefits that are currently desired for the targeted system (security, compatibility, resilience, adaptive optimization). From this set-up, a series of principles was deduced, i.e. in order to maintain and ascertain the validity of each virtualization in place we have to:

1. **Tighten boundaries** – Ascertaining that all of the resources which are being virtualized are explicitly treated as virtualizations, and thus not being virtualized redundantly. If they were, it is likely they inadvertently encompass parts that were neither desired nor taken into account at a latter stage when applying the other principles.

2. **Reinforce borders** – Identifying the relevant interfaces, protocols and conventions that connect the activity in the virtual space to its machine space.
Thereafter preventing, detecting and removing the interactions (i.e. bidirectional data-flows across these borders) that rely on, or take advantage of, lax, unintended or otherwise ambiguously specified data-flows.

3. Act on anomalies – Placing reactive measures that deal with undesired interactions as identified by Principle 2. Integrating these measures with the implementation of the virtualization, ascertaining that these can be activated not only by the intended monitoring conditions, but also by a trusted source (any person or external process with the means and authority to modify the virtual space of the subject).

4. Implement Monitoring – Sampling, gathering and presenting behavioral data from a. the virtual space, b. the virtualization and c. the environment, in order to evaluate the influence on resilience and the validity of protective measures and to empower involved stake-holders by providing representative information as to the dynamic properties and overall status of the subject at hand.

These principles combined assist in hardening the use of virtualization as a form of encapsulation of a subsystem and should be applied iteratively, essentially allowing for adaptive dynamic hardening. The success of this relies on establishing the external state holders that influence and hinder controlled experimentation with a subject (Fig. 9.1). If such state holders are not accounted for (4) and are allowed to persist as time progress, the end system will grow to encompass both, reducing or eliminating whatever virtualization benefits that was initially desired, and then the subsystem can not legitimately be considered decoupled. The remaining option then would be to virtualize the external state holders separately, with a possibly different translation mechanism, increasing overall complexity.

Finally, as a step towards experimentally validating these principles, in the context of critical software-intensive infrastructures, a distributed ICT experimental environment (Fig. 9.2), EXPII (Chapters. 4, 6, 8) was engineered following the research question:

- RQ4, What core services and components are needed to construct experiment environments capable of experimenting with the resilience of software-intensive critical infrastructures?

Note that, in addition to the SCADA, a number of other components – Fault Recorder, Fault Path Indicator and Agents are introduced. These are described in more detail in Chap. 6 and are part of a suggested extension to current SCADA for introducing self-healing (detecting, localizing and correcting problems) in the grid, as a resilience mechanism.

The remainder of this section will look at how this environment map to the aforementioned principles from the perspective of resiliences prerequisites (decoupled, self-healing, hardened, monitored).
Decouple Components

Decoupling is a fiendishly simple idea. You can have systems that are somehow artificially strapped together. The task is then to simply find these bonds and remove them, and somehow the situation has improved. On the other hand, we can examine the method by the way of two simple analogies. For instance, we can use nuts, nails, bolts and welded joints to piece together the raw materials of a structure and, except when restructuring or salvaging materials, it seems foolish to even try and remove these as a means of improving the structure. However, as in the case of conjoined twins, there is more interest and value in being able to separate the two, even if this is not a particularly easy task which always comes with a high risk.

With software, maintaining low coupling is an often desired design-time value, but when the software in its usable form has finally been put together, it is futile and rarely possible to arbitrarily remove any distinguishable part; code is data, but data is also code. In that sense, decoupling is used more as a metaphor than something which is finally engineered. What this metaphor establishes in the current context is essentially which relative parts that are external state holders and which parts that can be located within a virtual space. In more practical detail, software is seldom entirely monolithic. While operating system kernels are a commonly used example of software monoliths, they can still be affected by loadable device drivers. Similarly, software does not consist of loosely connected small parts that can be grabbed of a shelf, and glued together.

The virtualization parallel to decoupling components is in part the establish perimeter principle, and in part the reinforce protocols principle. The perimeter that can be established, however, can be arbitrarily selected by some stakeholder. When that has been said and done, reinforcing the protocols involved can be viewed as a preparatory means to enable restructuring, which would only ever be a safe operation with a subsystem that fulfills the virtualization ideal.
9.3. Restructuring SCADA

Connecting this principle to the demonstrator, the perimeter and the decoupling are established based on the observation that the involved components had been coupled with the specific networking environment in which they were developed. For instance, the communication between the fault-passage indicators and the fault-recorders was not intended to use other parts of the SCADA, nor were the agents supposed to communicate directly with the FPIs. The links between each component, furthermore, needed to be sized and work in separate networks or subnetworks; conditions that were not in effect when the individual components were developed and integrated.

Implement Self-healing

Like the case with decoupling, self-healing also appears to be a fairly simple task: You only need to have a part of the system detect errors, localize the underlying fault and then apply corrective measures. However, when trying to implement this in software, it rapidly becomes obvious that this is difficult if not outright impossible. While error detection is a direct effect of the reinforced protocols, localizing the underlying fault is not. Returning to the classification scheme in Sect. 2.4, which stipulates the effects that are observable during execution, i.e. data corruption, terminal state and inadequate performance, we note that not all of these can be readily detected and, furthermore, that they can be causally linked and cascade. Since the data or state relevant to untangle such effects can be irreversibly lost very rapidly under these circumstances, we cannot reliably reverse the chain of events back to the initial cause.

A Suggested solution to this predicament [73] is to exploit the possibility of repeating the computing performed between the last snapshot and the observed effect. This is achieved by generating a test-case\(^2\) that makes the fault reproducible, and then enumerate the space of possible interactions until a relevant subset of causally relevant contributors can be determined, something that may require hundreds of thousand of repetitions. This is entirely unfeasible for critical software-intensive infrastructures.

Since models for self-healing have progressed a lot further with respect to the powergrid, network communication and similar endeavors, and that one of the main points of the demonstrator was to illustrate viability from a power grid perspective we leave the notion of self-healing software outside the scope of the thesis.

Iteratively Harden

A hardening software system has at least two distinct, but complementary, perspectives. Typically, the most commonly used one concerns the removal of services or processes that are deemed superfluous and which come into effect with default configurations of larger pre-packaged software such as operating system distributions, where many services that could in some general sense be considered useful or interesting, are in fact irrelevant or insecure for a specific setting.

\(^2\)In execution, this concerns storing and replaying all interaction that occurs within the timeframe from the last accessible snapshot and the detected error.
The other perspective is, in essence, repairing (debugging) problems in one specific instance and, if possible, generalize it to other instances. This corresponds with the virtualization principles of reinforce borders but also act on anomalies, even though it is not the main intent behind said principles but rather a subsidiary effect of combining the two.

This principle was applied during the course of development of the demonstrator and the subsequent experiments, in the sense of individual filter configurations (firewall rulesets) on the nodes governing the subnetworks. During the initial runs, most of the default services and associated communications were allowed through, and from post-mortem analysis of network traffic, these filters were reconfigured to only allow traffic that was then known to be needed for the normal and self-healing operation of the SCADA and the agents.

**Introduce Monitoring**

The last principle, is shared by both the resilience and the virtualization perspectives and concerns gathering and presenting data about the internal states and interactions of a system, rather than its distinct inputs and outputs (directed towards the users and operators of the system). Hence, the difference is primarily that of stakeholders and demarcation.

From the virtualization standpoint, monitoring is needed for external error detection but also in order to find reconfigurations that could invalidate previous efforts.

The monitoring used (Fig. 9.3) for the demonstrator was partly the HMI of the SCADA as such. This was used to verify the function of the agents and of the fault-injection in the microgrid and so on, but also for verifying the transparency of the virtualization provided by the experiment environment. At the same time, however, these inputs and outputs would, from the perspective of the experiment environment, be regarded as internal states.

The monitoring was therefore further complemented by having the routing nodes continuously logging all traffic that was passed through each subnet and gen-
erating real-time graphs describing the number of packages and the amount of traffic (custom scripts). The last piece of monitoring was provided by having the controller repeatedly sending out latency probes to the nodes (using Smokeping [137]).

The traffic logs enabled post-mortem analysis when combined with tools for that purpose (the results presented are from the use of Wireshark [138]). The graphs from the individual nodes provided an internal dynamic view of current activities, and the trace probes provided an external dynamic view. These were all combined in an administrative web-interface for the experiment environment as such.

9.4 ICT for Smarter Grids

The previous section explained the foundation for the experiment environment built around a preexisting SCADA and smart grid prototype as part of a demonstrator for self-healing microgrids. This section expands on RQ4 combined with:

- RQ5– Which scenarios would be useful for both experimentation, training and for generating input to monitoring tools and tuning protective measures?

First, recall that the primary purpose of the demonstrator was the feasibility of self-healing microgrids during a variety of undesired conditions and the experiment environment provided partial state control (rollback for repeating experiments etc.) and different controlled modes of operation ranging from fault injection in the network communication (latency, packet loss, variations in bandwidth) to varying degrees of compromise by a third party (remote monitoring to remote tampering).

Among the secondary purposes was to gather first hand data as to the detailed behaviors of the solution as such, to be used as input for future iterations and refinement to the environment, assist in prioritizing etc. The initial ICT monitoring setup was purposefully restricted to generic, free and open solutions.

The final environment ended up similar to the one illustrated in Fig. 9.2, but expanded to have four physical nodes and four corresponding subnets (FPI, FR, Agent, SCADA) after it was discovered that the initial configuration of 3:3 had a side channel (external state-holder) into the SCADA system due to networking constraints in the university LAN (some of the components of the full SCADA system were also used for other labs, demos and projects).

Fig. 9.4 depicts the latencies of the SCADA subnet over the course of a day of experimentation. For more detailed descriptions of the experimentation as such, please refer to 8.7.2.

Mode 3 corresponds to remote tampering, meaning that the traffic from every subnet was redirected from the lab in Grenoble to the one at Blekinge Institute of Technology. The colored lines indicate the average latency, the color shows the packet loss experienced and the different shades of gray indicate the variance between probes.
Figure 9.4: Latency traces for several iterations of a self-healing sequence.

Figure 9.5: Protocols (Ambient, Modbus, RPC/DCOM, Other) in proportion to the total traffic of each subnet.

Fig. 9.5 depicts the distribution of the detected protocols in each subnet relative to the total amount of traffic within that subnet, extracted from the snapshots of the raw traffic logs (libpcap [139] format). Note that traffic marked as ambient concerns traffic that was identifiable as part of the upkeep of the devices in the network as such and thus generic to these devices’s respective operating systems (ARP, NTP, Samba and SNMP/STP from switches). The traffic generated by the ping-probes from the controller was filtered and excluded from all graphs. In the case of the FPIs, the increase in ambient traffic is due to the ARP MAC - IP discovery / refresh. Even so, it still is notably high.

Furthermore, the traffic in the Agent subnet marked as Other is, on closer inspection, also RPC/DCOM that could not reliably be detected as such. The most probable explanation to this fact is that the implementation of the protocol deviated slightly from what was expected by the dissector module in the analysis tools and, subsequently, the implementation lost track of the dynamic port allocations that are central in the design of this particular protocol. This explanation is supported when repeating the analysis on larger dumps where the proportion of data that could not be properly classified grows as time goes by, even though the exact split point where the stream goes from detected to undetected varied.

Table 9.4 shows the proportions of the total measured traffic that passed between the different subnets. Note that this only relates to the traffic necessary for fault detection and self-healing, and not to other kinds of SCADA traffic. Other activities of an operator that is confined to the SCADA as such are not covered.
9.4. ICT for Smarter Grids

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>% of total traffic</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPI</td>
<td>FR</td>
<td>22</td>
</tr>
<tr>
<td>FR</td>
<td>FPI</td>
<td>25</td>
</tr>
<tr>
<td>FR</td>
<td>Agent</td>
<td>26</td>
</tr>
<tr>
<td>Agent</td>
<td>FR</td>
<td>5</td>
</tr>
<tr>
<td>Agent</td>
<td>SCADA</td>
<td>10</td>
</tr>
<tr>
<td>SCADA</td>
<td>Agent</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 9.1: Traffic ratios.

<table>
<thead>
<tr>
<th>FPI</th>
<th>FR</th>
<th>Agent</th>
<th>Scada</th>
<th>Time Elapsed</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0) (3/6)</td>
<td>(0) (10/167)</td>
<td>(0) (130/21)</td>
<td>(0) (25/25)</td>
<td>6s</td>
</tr>
<tr>
<td>x (3/2)</td>
<td>x (2/0)</td>
<td>x (20/1)</td>
<td>x (3/3)</td>
<td>36s</td>
</tr>
</tbody>
</table>

Table 9.2: Notable link configurations (latency introduced, milliseconds), peak traffic rates (output/input) KiB/s and time for the self-healing scenario to complete.

Table 9.4 depicts some notable values from a series of iterations of the same self-healing scenario and the time elapsed from the point where a fault was detected to the point where an agent tells an operator which breakers to open and/or close. As a point of reference, the upper response time (depending on country, local regulations, fault-type and other factors) is around 160s.

Then, Fig. 9.6 shows one successful run of the self-healing scenario, starting from a forced reset of the monitoring. It is quite clear, even without access to domain expertise and using only course-grained measurements, where and when the process is the most vulnerable.

To summarize, some of the major key points noted have been:

- The communication between the fault-recorder and the agent is the most influential. Should such a solution be integrated into current grids, the fault recorder would be part of the RTU in a transformer station, while the agent solution would be close to the HMI. In this case, even small increments in latency between Agent and FR lead to drastic increases in the time required for the self-healing process.

- It is trivial for a third party to detect when self-healing is initiated just from the anomaly in traffic patterns. This moment is also when the system is most vulnerable. Even a denial of service attack on the communication infrastructure would suffice to increase the damage done to the grid, suggesting that the monitoring infrastructure should indeed be able to inform the operators about current communication network conditions.

- The higher level communication protocols (e.g. DCOM) involved are poor choices in respect to establishing protective devices within the confines of
9. Results

Figure 9.6: Annotated graph of one iteration of a self-healing scenario, from the perspective of the node governing the agent. Thus, output entails data being sent into the agent subnet.

The developed environment currently enables us to virtualize, monitor and intervene with, among other things, the communications of a SCADA-system governing a micro-grid cell from the perspective of either an analyst, an antagonist that have gained link-level access, or that of an antagonist who has breached or circumvented higher barrier security measures (VPN-tunnels, firewalls, Intrusion Detection Systems, etc).

Output and lessons learned include the suggestion to devise training scenarios based on real-world events and antagonistic threats, categorized into Business Intelligence, Sabotage and finally, Business Sabotage, as exemplified in (8.3) and subsequently use these scenarios as means for developing and improving protective measures.
9.5. Framework for Systemic Debugging

In lieu of doing this on a live-SCADA system, we can introduce multi-agent systems into our experiment environments, as means for enforcing specific systemic state transitions, given a range of domain-specific preconditions (as in the case of the demonstrator, detecting a failure and taking a self-healing action) and thus generate realistic ambient data as an adaptive and dynamic alternative to replay driven approaches like TCPOpera [62] or mirroring live SCADA activity, even in cases where replay driven approaches would fail (external and/or persistent state-holders that were unaccounted for).

We suggest two overarching attack patterns for deriving more detailed scenarios. The first one is that of RogueSCADA which is an attack in the data-domain, representing an alternative take on the common threat of an attacker gaining partial access to a subset of the SCADA. In contrast to the Stuxnet [101] case however, where propagation was primarily inwards toward a directed subtarget (PLCs connected to centrifuges), RogueSCADA would be directed outwards from sensors to convincing operators to take action based on faulty data. This is motivated in part by the large geographical distribution typical of the power grid, where breaches in remote transformer stations would be a more subtle and plausible point of entry than a breach at a control center, but also in part due to the close match between this pattern and the capabilities of the remote monitoring and remote tampering modes of operation in the demonstrator.

The other attack pattern is that of the Empowered User and stems from the compromise of partially trusted devices placed on a massive scale in untrusted environments, both in the form of the smart meters themselves, but also in the form of the miniature SCADAs-like systems that emerge from smart home initiatives. Due to the high number of instances (homogeneity), it is plausible that an exploitable vulnerability affecting one device will be able to affect several others in its vicinity, as have been shown in similar devices (smart-phones and gaming consoles). These can be leveraged to gain economic advantages by manipulating shared energy markets, surreptitiously damage businesses and the grid itself (provoking protective devices to go off in patterns similar to that of previous blackouts).

9.5 Framework for Systemic Debugging

The previous section covered how the ICT monitoring was implemented in the demonstrator, along with the first round of findings, showing some of the problems belonging to a planar or “sliced” view of the ICT, suggesting that an interleaved or “hybrid” monitoring approach would be advantageous, along with details on scenarios that could prove useful in developing and testing such monitoring.

For this section, we expand on the method aspects (RQ5) and combine them with a more technical software approach as to selecting the set of tools (RQ3) that could be used in combination with the aforementioned scenarios.
9. Results

- RQ3, Among the sets of tools that enable dynamic instrumentation of software-intensive systems, which are suitable for controlling and monitoring virtualizations in critical software-intensive infrastructures?

Since we had limited access to the demonstrator as a whole, we looked for alternative environments to use as an analog for experimentation and finally settled on late-stage or systemic debugging in smart-phones, sharing many properties that overlap with the vision of monitoring the smart grid. Some of these properties include that smart phones are mostly comprised of commercial, off-the-shelf components (COTS) ranging from the larger “platform” (pre-packaged combinations of everything from modem to supporting Digital Signal Processors (DSPs) to main CPUs) all the way to the operating system and supportive software and it is rather the integration, optimization and customization that are the key handset vendor contributions. A key difference however, is the massive involvement of small third parties that comprise a notable portion of the executing software. This fact, combined with volumes in the range of millions of devices, makes for a comparably aggressive development environment where legitimate software developers are openly encouraged to employ surreptitious, anti-debugging techniques, for piracy concerns, if nothing else.

We worked with a debugging task-force, MiB, at Sony Mobile (formerly Sony-Ericsson). A task-force comprised almost exclusively of senior engineers specializing in embedded systems. Among the goals of this cooperation was to gather data as to where the cutting edge of tools and methods was in regards to instrumenting and manipulating large and unwieldy software-intensive systems, which mechanisms they rely on and what the downside of using these tools could be. This started off with the co-development of a course in systemic software debugging used as both a local course at the company, and as a computer science masters level course at BTH. Most of the developed reading material has been made freely available online [70]. Although the core of the course contents was shared between the two courses, the one that was used at Sony had a seminar setting of smaller groups (around five participants per group) with individual assignments being based on work-related topics, while-as the course at BTH was based on lectures with a larger audience and examined solely based on laboratory assignments.

Some of the major shortcomings of the tools employed that were noted during the seminars (a more detailed description can be found in Sect. 7.2.1) are as follows:

1. With the symbolic debugger, the level of control and intervention required for implementing breakpoints\(^3\), which are necessary to maintain source-level symbolic debugging will be increasingly difficult to achieve for some stakeholders. Furthermore, the interfaces used to achieve such control are quite easy to detect from the targeted code, and many legitimate (copy and integrity protection schemes and other forms of surreptitious software) and dubious (worms, viruses) programs take advantage of this fact to alter their behavior in, for the analyst, counter-productive ways.

\(^3\)This is a central control mechanism in that it specifies a location in memory that, when executed or modified, will transfer control to a handler routine.
2. With post-mortem analyzers, the relevance of the accessible information can be very high when the effect studied falls within a proximate onset, proximate cause kind of scenario. However, relevance quickly shrinks with time as the state holders required to perform the analysis get overwritten at a rapid rate. Thus, post-mortem analysis depends on how much of the type and location of specific data that can be determined in advance, but also on the trigger that induces a terminal state or generates a snapshot is proximate in time to the underlying issue.

3. With tracers, the central issue is the model to which gathered measurements are attached. Even though this modeling challenge is shared with post-mortem analyzers to some degree, a key difference is in the source of the measurements and the relative timing. In the post-mortem case, the concern is how much data can be extracted and made useful from an instance that has come to a very distinct halt. In the case of tracing, you rather have a series of tools that provides small samples of a few specific key data, often generated throughout the life-span of the program in question. The most primitive of these tools are the ones that are integrated in the subject, e.g. printf statements or calls to system log facilities.

![Figure 9.7: Simplified view of interaction between tools and organizational units.](image-url)

Another key point was the importance of how the tools were combined and configured to interact with the larger organization (Fig. 9.7). Of particular note was the use of an in-house post-mortem analyzer that could generate highly detailed reports based on a snapshot or crash dump. These reports acted as “living documents” of sorts; small enough to be passed around, detailed enough to be useful in figuring out the underlying cause and little training was needed to learn how to use the tool to generate reports. However, with the increasing shift towards less control on behalf of the vendor in regards to the end-composition of the software configuration in use on any single instance, the precision and subsequent usefulness of such reports decrease rapidly.

A recent development in debugging tools is that of tracing frameworks. While developing a trace-tool for gathering highly specific measurements when provided with access to source code or target-specific knowledge is trivial, especially when compared to something more delicate, like a symbolic source-level debugger. Unfortunately, the utility of these are fairly limited; they are typically bound
to one sampling mechanism and one form of representation that don’t lend themselves to combination or coordination. To improve re-use and allow tracing across system borders, tracing frameworks like Dtrace [52] have been developed and are slowly making their way into the mainstream, particularly in diagnostics and system administrative tasks. They are typically comprised of some notation format for configuration, a program for coordination and control (Fig. 9.8), an API for defining probes (Fig. 9.9) and a set of pre-integrated probes.

As noted in the context of other tracing related frameworks, e.g. Xtrace [63] in addition to the principal problem of a debugger having to be truthful [50], is the difficulty in finding representative evaluation scenarios, limitations in presentable topologies (representation) and the need for heavy integration into the subject. Given the constraints from the previous sections in the chapter, means that we need to look elsewhere for solutions.

To this end, we have developed a framework for systemic tracing (Fig. 9.11) intended to be used both as a monitoring system for the state and progress of the experiment environment as such, but also as an exploration tool for creating customized monitoring models for the experiment subject itself and as a tracing aid for systemic debugging purposes. This framework uses a programmable networks of dynamic probes with configurable sampling mechanisms (ranging from exploitation of buffer overflows and function interposing to operating system provided tracing and debugging interfaces) mapped to interchangeable non-native representations.
9.5. Framework for Systemic Debugging

Figure 9.10: Dynamic Adaptive Monitoring.

Please refer to (7.4) for more detail and Appendix B for a sample script illustrating its use, with complete source code of the project available at [109].

Figure 9.11: Overview of the Arcan/SenseEye tool.

In order to tackle the aforementioned challenges, demonstrate the viability of the approach and generate as large a test surface as possible within a short time-frame and a limited budget, we branched the framework into two separate tools that target alternate communities. The tool covered here, Arcan, is focused on real-time visualization, similar in capabilities to modern game engines, but with security and instrumentation of non-cooperative subjects in mind.

The first probe that was developed is capable of surreptitiously intercepting and redirecting parts of the rendering stages in some emulators, and, optionally, programmatically inject input events into the event loop of the subject.  

4The other tool, SenseEye is currently more of a sandbox for notation experiments, see Future Work, Chapter 10.
9. RESULTS

Among the reasons for choosing emulators, and particularly emulators targeting legacy computer hardware running video games as a first subject for experimentation, we find:

- **Virtualization Ideal**, A prime example of using whole-system virtualization for compatibility.
- **Resource Use**, Interpreted emulation is notably CPU intensive with a high frequency of invocation and a high translation cost.
- **Strict Deadlines**, Set to run at a fixed refresh rate.
- **Data Intensive**, Sampling hundreds of megabytes of data per second may be required.

In addition, due to the visual and interactive nature of the games themselves, the observer effect imposed by the probe can be readily evaluated both qualitatively and quantitatively, and regular video editing tools can be used for recording, comparing and replaying consecutive runs as automated means for finding regressions (compare instrumented iterations with non-instrumented ones).

The probe demonstrated (7.4.1) successful interception and synchronous sampling without contributing adversely to jitter in video refresh deadline alignment at low resolutions, and, asynchronously, at higher resolutions – assuming sufficient bandwidth is available.

In closing, Section 9.3 provides mechanisms, restrictions and method. Section 9.4 provides the environment, context of use and validation scenarios. Lastly, Section 9.5 provides the tools, training and further validation. These sections combined, provides us with the means for **Monitoring Infrastructure Affordances**.
10 Discussion

This chapter takes a more personal and reflective stance on the contributions made, as means for providing additional context and comment outside the direct scope of the thesis as such. The sections are as follows: In Related Work, some specific external projects and research interests are covered, particularly in relation to the Results chapter. In Feedback, the general reception that some of the ideas presented herein received at conferences and other forms of dissemination is summarized and commented upon. In Criticism, some notable flaws, shortcomings and limitations are laid out. The Future Work section suggests possible future use and development of the presented contributions, rounding things off with the final section, Concluding Remarks.

10.1 Related Work

Although directly related work has been referenced as such in their appropriate contexts, there are a few on-going efforts that should be explicitly mentioned and considered with Chap. 9 as reference.

10.1.1 NSF GENI

GENI [64] is mentioned briefly in Chap. 4, and is a U.S. National Science Foundation sponsored project aimed at providing the infrastructure and support for international virtual laboratories hosting experiments for future networks at scale.

![Figure 10.1: Architecture of the NSF GENI experiment platform.](image-url)

141
Revisiting the GENI architecture (Fig. 10.1) one of the connected aggregates in which a researcher can allocate resources is PlanetLab [44], as a research platform for future Internets, Overlay Networks etc.

A possible contact point here is thus the option of connecting the remote monitoring (mode 2) or remote tampering (mode 3) side of the EXP-II environment (Chap. 6) to a cluster of allocated nodes in PlanetLab as a first development step towards experimentally validating monitoring and defense for the Empowered User (Chap. 9.4) attack pattern.

10.1.2 National SCADA Test Bench

For a “big-iron” facility in the SCADA domain, the U.S. National SCADA Test Bench (NSTB) [140] is a prime candidate.

As per their facts sheet:

The NSTB is a resource for identifying and solving today’s SCADA vulnerability issues, testing new and existing equipment, and developing next-generation architectures and technology advances.

Primary goals are to accomplish the following:

- Raise industry awareness of system vulnerability issues and mitigation techniques.
- Collaborate with industry to identify, assess, and mitigate current SCADA system vulnerabilities.
- Work with industry to develop near-term solutions and risk mitigation strategies for existing systems.
- Develop best practices as well as next-generation architectures for intelligent, inherently secure and dependable control systems and infrastructures.
- Support development of national standards and guidelines for more secure control systems.

Albeit that much of the facilities involved are sealed off from the public eye for national security reasons, there is still considerable international impact from cooperation with multinational vendors, influence on standardization and training material through their industry outreach program. A possible contact point here is thus looking at combining the type of training material in use for security teams, with that of debugging and the more domain specific one of day to day power-grid operations.

10.1.3 Ultra-Large-Scale Systems

While some chapters of the thesis paint a fairly grim picture of the state of debugging in general, and systemic debugging in particular, in respect to current Software Engineering efforts, the problem picture painted herein is shared to a large degree with that of ULSS [60].
Although they are being described as a thing of the future, we would argue that it is not only a thing of the present, but have been so for a while, and that the subjects studied herein are prime examples of that.

To quote their research agenda:

*Fundamental gaps in our current understanding of software and software development at the scale of ULSS systems present profound impediments to the technically and economically effective achievement of information superiority. These gaps are strategic, not tactical. They are unlikely to be addressed adequately by incremental research within established categories. Rather, we require a broad new conception of both the nature of such systems and new ideas for how to develop them. We will need to look at them differently, not just as systems or systems of systems, but as socio-technical ecosystems. We will face fundamental challenges in the design and evolution, orchestration and control, and monitoring and assessment of ULSS. These challenges require breakthrough research.*

This fits remarkably well into both the scope of the NSTB and of the Global Environment for Network Innovations (GENI) projects and there is a possible contact point in the future work (see below) of the tools developed herein, particularly with respect to monitoring through dynamic tracing and as a context for evaluating such notations.

### 10.2 Feedback

With contributions being submitted and accepted (or rejected) for publication, there is usually some valuable feedback that inevitably is lost unless it can be retrofitted into revised submissions or explicitly be taken into account in future ones, but the thesis format provides an additional opportunity to do so. While the reception has been favorable in the selected industrial and security-related forums, it is always useful to take alternative ones into account in order to reduce confirmation bias. Among the stronger and representative critical opinions delivered from a software engineering perspective in regards to the specific approach of engineering resilience through systemic debugging is well-summarized by this quote:

>“You make a very brief mention of formal methods such as model checking. It seems to me that this is at least part of the overall solution. I’d like to see more on all of the techniques that are useful in this domain, and then zero on the ones you have decided to pursue.”.

It is easy to brush this off with the opposite opinion, namely that so much ink has been spilled on that particular subject [88] that any alternative would be a welcomed breeze of fresh air by now. The generalized debate is an ongoing quarrel in computing\(^1\) that both needs and deserves considerable more space for a structured dissection of the pros and cons involved than can be provided for here, but ultimately, we do not consider these methods as much more than very distantly related to the work in this domain. During optimal conditions, formal methods may well transform any resilience mechanisms in place into redundant safeguards. Until such a time however, it is the other way around – resilience mechanisms are in place to account for all the inadequacies of formal methods.

---

\(^1\)For examples as to the tone of the debate as such, please refer to [65] [66] [67]
The very idea of retrofitting these systems with yet another model in a formal notation (not forgetting the one that the underlying programs were written in) which still needs to be developed and verified against the original requirements (assuming that these requirements are representative and accessible), introducing and managing yet another tool in the chain from specification to execution (in order to enforce these models, and if they’re not enforced, how should any deviations be handled?) seems quite risky. In addition, if we concede on the notion that these techniques can be retrofitted, and only use it for future software to integrate, what should we do about everything that is already up and running?

While alternative suggestions such as the introduction of more resilience mechanisms may indeed be a case of putting yet another band-aid on a wound that, in hindsight, should have been surgically sutured from the start, it is easy to forget that these mechanisms are largely what keeps current systems up and running, or just how much protection we actively rely on and is indeed offered by the likes of virtualized memory enforced process separation, address space layout randomization, stack cookies, journaling file-systems and so on.

10.3 Criticism

Given the number of years of work behind the average Ph.D Thesis, there are undoubtedly things that, at least in hindsight, should have been approached differently which would quite possibly have resulted in, perhaps not better results, but at the very least a more clear and concise presentation. This section will therefore present two such lessons learned.

The first one is that debugging and software security well represents the pink-elephant in the room, in the sense of that one big eye-catcher that everyone notices yet still likes to pretend is not there. Debugging and Software Security are at the same time in conflict and in cooperation. The tools used to expose, localize and correct defects operate on the same mechanisms that the tools used to develop, or to use an ugly rhetoric, weaponize, an exploit for a vulnerability, and can in many cases can be used interchangeably. On the other hand, a well-written exploit perfectly illustrates how a seemingly insignificant inconsideration can be taken advantage of in order to controllably and repeatably re-purpose a running program to do something that the designers and developers did not intend. There must surely be something in that which can be harnessed for more productive use. For the majority of this thesis, we have been operating in the intersection between the two, yet the terminology employed have been primarily that of the debugging perspective rather than the security one. Instead of surveillance, we say monitoring, instead of forensics we say crash dump analysis and so on. It would probably have been for the better if the relationship between Debugging and Software Security endeavors had been mapped out from the beginning rather than being implied and scattered.

The other one is on the subject of the use of analogical models as a basis for experimentation and explanation. This problem is, in a sense, connected to the oversimplified SCADA view in use (Chap. 2.3.1) allowing for some quite generous generalizations. The design and implementations of training scenarios for the
exemplified attack patterns would likely have gone further with a layered end-to-end model taking Smart Meters and data exchange between microgrid cells into account, especially in regards to the operators as they still have to withstand social engineering techniques as part of the larger attack.

The bigger analogies in place was the use of the software analyst in place of the SCADA operator, the smart phone setting as a possible model for future smart meter monitoring and ULSS, and the emulator as an example for whole-system virtualization for compatibility (running software developed for dated hardware on modern hardware with an enabling run-time translation layer, the emulator). For the SCADA operator, the issue is the unspecified role of prospective future smart grid operators and whether that should include the empowered user or not. While using the emulator analogy as an observer-effect measurement in dynamic instrumentation evaluation seems reasonably safe, assessing how accurately an emulator emulates the behavior of the targeted system outside of a cognitive model with a known reference (e.g. a video game), is difficult even for trivial targets.

**10.4 Future Work**

The previous section covered possible improvements to the work done and can therefore, in some sense of the word, be considered possible future work if the topic would be revisited. This section however, is directly focused on how some of the contributions can be expanded upon.

For the developed tools Arcan/SenseEye (Chap. 7), the set of features that have already been implemented seem sufficiently complete for them to move on towards being applied on more sensitive subjects, with experiment environment deployment- active state- and realtime resource usage- monitoring being prime targets. But, given the complexity of the tools and the relatively short time they have been in active use, the responsible way forward seem to be to let the codebase stable and harden (as per the virtualization principles of Chap. 5) while improving the softer sides e.g. API documentation and examples and more thoroughly select and improve on the available selection of non-native representations.

For the experiment environment (Chap. 6) in question, the ambition has been to make the supporting tools used to build EXP-II as accessible to the general public as possible – in an open-source form much akin to the one used on Arcan. During investigations towards this goal however, it became clear that while the underlying ideas, the selection of services and overall architecture has withstood the test of time, development in adjacent areas, especially with regards to software and hardware dependencies has changed to the degree that the best option is to use the current solution as a source of inspiration rather than something concrete to refine and develop further.

While the experiment environment is still a valuable counterpoint to the big-iron ones that can be constructed through cloud-like services, the best option seems to be to push for a “more with less” approach taking advantage of the rapid development and availability of cheap and energy efficient System-On-A-Chip (SOC)
10. DISCUSSION

Figure 10.2: Cover of Systemic Software Debugging.

ARM-based platforms. In fact, some such platforms have been demonstrated [68] as a cost-efficient way of emulating field devices (PLCs) in industrial control systems (virtualization for compatibility).

For the larger picture, i.e. combining the methods, tools, and environments provided herein into a larger framework, however, it seems like the best opportunities are still several years away from being realized in a critical setting, which, in a sense, is a good thing.

10.5 Concluding Remarks

In active multidisciplinary fields of research, such as the one we have been working in, it is quite possible that the best of solutions lie not in the paths that one has chosen to pursue, but in something else entirely, and with computing and critical infrastructure in particular, the allure of digitalization as a cheap and modern way out may well not be the right one. The last figure therefore, Fig. 10.2, is from the cover of Systemic Software Debugging, used here as a light-hearted way to highlight two key points.

The first point concerns the issue of building very-large, software-intensive systems out of re-purposed commercially available components picked off the shelf. Even if the components themselves have been formally verified and validated according to some preset requirements and, according to some quality evaluation standard, shown to be of good quality, flexible interfaces can yield unexpected machines. Given time and a reactive stance attacking problems as they arrive, the end result may well be an overly complicated solution for solving something fundamentally trivial. While the example in the figure is complete in the sense that all components that are needed in order to understand what the machine is supposed to do are present, occluding just parts would notably increase the difficulty in figure that out.
With software, on the other hand, we still lack many of the means and faculties necessary to reliably observe software execution in any meaningful way; we can observe the inputs and outputs that the machine operates on, but now what it actually does.

The second point concerns the power of various forms of causal notations for modelling intended or desired behaviors in complex systems, be it sheet music, some programming language currently in fashion or, as shown in the figure, a “Rube Goldberg” machine. Useful such notations, enables the professional, in part thanks to the experiences that makes him or her an expert, to make well-grounded predictions as to the behavior of the modeled system, and thus, establish where and why things did not go according to expectations. Programming languages works on the basis of a well-defined starting state and describe in great detail how and under which conditions this state is to be modified. While these can surely be made to work from the other end, where system borders and current state is uncertain or unknown and still be able to present some plausible view of what is going on, there must surely be other forms of notations more useful in this regard.

In regards to the other principal topic, the implications of current and future threats to infrastructures, in the sense of something direct and concrete that we would like to be able to observe, monitor and respond to during execution, it is worthwhile to recall that the vast majority of people out there are fighting the good fight. There is, for now, an abundance of easier attack targets out there with a more favorable risk / reward ratio. The larger longer-term threat, by far, is rather that of well-meaning yet misguided attempts at removing- or restricting-access to the very same tools that enables these people to do what they do, through the use of policies, laws, certification processes, licensing regulations, and ultimately, force.

To end on a somewhat cryptic note using a slightly modified proverb; The early-bird “gets” the worm.
A  Glossary

This appendix section provides a list of abbreviations along with references to explanations of major terms used throughout the thesis.

### Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ARP</td>
<td>Address Resolution Protocol</td>
</tr>
<tr>
<td>ASLR</td>
<td>Address-Space Layout Randomization</td>
</tr>
<tr>
<td>CISC</td>
<td>Complex Instruction Set Computer</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial, Off-the-Self</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DCOM</td>
<td>Distributed Common Object Model</td>
</tr>
<tr>
<td>DER</td>
<td>Distributed Energy Resources</td>
</tr>
<tr>
<td>DG</td>
<td>Distributed Generation</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DMZ</td>
<td>Demilitarized Zone</td>
</tr>
<tr>
<td>DRM</td>
<td>Digital Rights Management</td>
</tr>
<tr>
<td>DSO</td>
<td>Distributed System/Service Operator</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>FPI</td>
<td>Fault Passage Indicator</td>
</tr>
<tr>
<td>FR</td>
<td>Fault-Recorder</td>
</tr>
<tr>
<td>GENI</td>
<td>Global Environment for Network Innovations</td>
</tr>
<tr>
<td>GNU</td>
<td>Gnu is Not Unix</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HLE</td>
<td>High-Level Emulation</td>
</tr>
<tr>
<td>HMI</td>
<td>Human Machine Interface</td>
</tr>
<tr>
<td>HV</td>
<td>High Voltage</td>
</tr>
<tr>
<td>ICT</td>
<td>Information and Communication Technologies</td>
</tr>
<tr>
<td>IDS</td>
<td>Intrusion Detection System</td>
</tr>
<tr>
<td>IED</td>
<td>Intelligent Electronic Device</td>
</tr>
<tr>
<td>IPS</td>
<td>Information Processing System</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IS</td>
<td>Information System</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>JNI</td>
<td>Java Native Interface</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>KPI</td>
<td>Key Performance Indicators</td>
</tr>
<tr>
<td>KVM</td>
<td>Keyboard, Video and Mouse</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LLVM</td>
<td>Low-level Virtual Machine</td>
</tr>
<tr>
<td>LSO</td>
<td>Local System/Service Operator</td>
</tr>
<tr>
<td>LV</td>
<td>Low Voltage</td>
</tr>
<tr>
<td>MAC</td>
<td>Media Access Control</td>
</tr>
</tbody>
</table>
## A. Glossary

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAME</td>
<td>Multiple Arcade Machine Emulator</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>MTU</td>
<td>Main Terminal Unit</td>
</tr>
<tr>
<td>MV</td>
<td>Medium Voltage</td>
</tr>
<tr>
<td>NAT</td>
<td>Network Address Translation</td>
</tr>
<tr>
<td>NOP</td>
<td>No OPeration</td>
</tr>
<tr>
<td>NSF</td>
<td>National Science Foundation</td>
</tr>
<tr>
<td>NSTB</td>
<td>National SCADA Test Bench</td>
</tr>
<tr>
<td>NTP</td>
<td>Network Time Protocol</td>
</tr>
<tr>
<td>OFS</td>
<td>OPC Factory Server</td>
</tr>
<tr>
<td>OSI</td>
<td>Open Systems Interconnection</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PLC</td>
<td>Programmable Logic Controller</td>
</tr>
<tr>
<td>PRNG</td>
<td>Pseudo- Random Number Generator</td>
</tr>
<tr>
<td>RAID</td>
<td>Redundant Array of Inexpensive Disks</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RES</td>
<td>Renewable Energy Sources</td>
</tr>
<tr>
<td>RFID</td>
<td>Radio Frequency Identifier</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>ROP</td>
<td>Return- Oriented Programming</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>RTU</td>
<td>Remote Terminal Unit</td>
</tr>
<tr>
<td>SCADA</td>
<td>System Control and Data Acquisition</td>
</tr>
<tr>
<td>SNMP</td>
<td>Simple Network Management Protocol</td>
</tr>
<tr>
<td>SG</td>
<td>Smart-Grid</td>
</tr>
<tr>
<td>SLA</td>
<td>Service-Level Agreement</td>
</tr>
<tr>
<td>SOA</td>
<td>Service-Oriented Architectures</td>
</tr>
<tr>
<td>SOC</td>
<td>System-On-A-Chip</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>STP</td>
<td>Spanning Tree Protocol</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>UNIX</td>
<td>Uniplexed Information and Computing System</td>
</tr>
<tr>
<td>ULSS</td>
<td>Ultra-Large Scale System</td>
</tr>
<tr>
<td>VA</td>
<td>Volt Ampere</td>
</tr>
<tr>
<td>VLAN</td>
<td>Virtual Local Area Network</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual Private Network</td>
</tr>
<tr>
<td>V</td>
<td>Voltage</td>
</tr>
<tr>
<td>WAN</td>
<td>Wide Area Network</td>
</tr>
<tr>
<td>WINE</td>
<td>Wine Is Not an Emulator</td>
</tr>
<tr>
<td>WORE</td>
<td>Write Once, Run Everywhere</td>
</tr>
</tbody>
</table>
Detailed Definitions

This section introduces complementary definitions key terms used in a more involved sense than standard dictionary definitions would permit, and points to specific parts of the thesis that attempts to explain relevant context and use. As mentioned in the Reading Guidelines, the reader is still expected to possess an understanding for basic computing terminology or to refer to [89] and secondarily [7] for additional guidance.

Virtualization

(Chap. 2.2, Chap. 5.1, Chap. 5.2) In its active form, virtualizing a resource (communication, storage, computation) means to insert or define a translation layer (interface) and make sure that whatever computing task (e.g. a running program) is accessing or is supposed to access this resource is doing that using this translation layer. This has the effect of decoupling the task from the resource in question and recoupling it to the interface. This means that, in theory, the underlying resource can be changed, replaced or otherwise manipulated without the task itself being affected, assuming that the translation layer is capable of hiding whatever is going on (being transparent).

When used as a noun, a virtualization refers to the implementation of this interface and is thus a possible embodiment of a computing abstraction. This implies that the translation layer is an active ongoing part of the end system and can itself be both observed and tampered with. In the special case of all computing resources being virtualized in the same translation layer, it is referred to as a virtual machine or whole-system virtualization.

Resilience

(Chap. 2.2, Chap. 4) This term is frequently used synonymously with fault tolerance both here and elsewhere as a means for referring to the ability of a system to handle partial component failures. There is, however, an additional dimension to the use of resilience for the ability to harness disturbances in a broader system of systems or socio-technical systems perspective, particularly within the framework of resilience engineering. For a deeper explanation of this term, please refer to [69].
The following example script demonstrate the use of Arcan in monitoring three different domains: physical (through a video capture device), network (through a local server to which other sensors can connect and push data) and a virtual (hijacking execution of an emulator), combining these into a single video stream stored locally (though RTMP is support).

```plaintext
function thdemo()
    cellw = math.floor(VRESW * 0.5);
    cellh = math.floor(VRESH * 0.5);
    reclist = {};
    srv = net_listen();
    resize_image(srv, cellw, cellh);
    show_image(srv);
    table.insert(reclist, srv);

    vcap = load_movie("vidcap:0");
    if (vcap and vcap ~= BADID) then
        resize_image(vcap, cellw, cellh);
        move_image(vcap, cellw, 0);
        show_image(vcap);
        table.insert(reclist, vcap);
    end

    glist = list_games({});
    if (glist and #glist > 0) then
        tgtid = launch_target(glist[1].gameid, LAUNCH_INTERNAL);
        resize_image(tgtid, cellw, cellh);
        move_image(tgtid, 0, cellh);
        show_image(tgtid);
        table.insert(reclist, tgtid);
    end

    codecopts = "noaudio:vbr=7:fps=60";
    outvid = fill_surface(VRESW, VRESH, 0, 0, 0, VRESW, VRESH);
    define_recordtarget(outvid, "output.mkv", codecopts, reclist, {});
    RENDERTARGET_NODETACH, RENDERTARGET_NOSCALE, -1);
end
```
c References
Articles


[34] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin, “Compatibility is not transparency: Vmm detection myths and realities,” Proceedings of the 11th USENIX workshop on Hot topics in operating systems, pp. 1–6, 2007. 5.4.1


[41] B. Schroeder and G. A. Gibson, “Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you?” in *Proceedings of the 5th USENIX conference on File and Storage Technologies*, 2007. 6.5.2


Books


[84] C. Cifuentes, Reverse Compilation Techniques, 1994, p. 56. 5.4.3

[85] I. Hacking, Representing and Intervening: Introductory Topics in the Philosophy of Natural Science. Cambridge University Press, nov 1983. 5.5.1


[87] S. Hussain, Coordination and Monitoring Services Based on Service Level Agreements in Smart Grids. Blekinge Institute of Technology, 2011. 9.1

Standards


[92] “Java native interface specification.” Available: http://download.oracle.com/javase/1.5.0/docs/guide/jni/spec/jniTOC.html 5.4.4


[99] I. E. Commission, ISA-99.02.01 / IEC 62443-2-1: Establishing an IACS Security Program. 9.1
Online Resources


ONLINE RESOURCES


ABSTRACT
Computing has made its way into most of our lives as a key processor of vast quantities of information. This has happened directly in terms of gadgets and devices that assists us in everyday life, but also indirectly, through the critical infrastructures that enables these devices to function. A key issue with critical infrastructures such as transportation, communication, power-grids and finance, is increasingly circular interdependencies. Because of this issue, a disruption in either one can cascade and have a global effect on the others. To manage these complexities, we are depending on a number of monitoring systems that allow operators and other stakeholders to, within their respective expert domains, discover disruptions as early as possible and then take appropriate actions.

These monitoring systems are not without challenges of their own. In addition to having evolved organically alongside their respective infrastructures, there is a considerable legacy to account for, with both hardware and software components spanning decades of computing history. This puts heavy restrictions on the kinds of interventions that can be performed safely, implying that these systems are ill fit for handling the software and software security landscapes of today, where updates and adjustments need to be applied on a daily basis in order to stand a fighting chance.

The work presented herein address some of the major challenges in securing these monitoring systems against current and future threats posed by antagonistic actors, dormant software defects and changes imposed by technological advances and academic discoveries. This is approached on several fronts in parallel: by embedding resilience in order to allow for controlled experimentation and evaluation of new protection mechanisms in incrementally sensitive settings; by developing laboratory facilities for resilient smart power-grids; and by developing tools and training scenarios for operators of adaptive and reconfigurable monitoring systems.