EXPLORING SOFTWARE RESILIENCE

Björn Ståhl

Blekinge Institute of Technology
Licentiate Dissertation Series No. 2011:05

School of Computing

Exploring Software Resilience

Björn Ståhl

Blekinge Institute of Technology Licentiate Dissertation Series
No 2011:05
ISSN 1650-2140
ISBN 978-91-7295-206-5

Exploring Software Resilience

Björn Ståhl

Computer and Communication Systems Research Laboratory
School of Computing
Blekinge Institute of Technology
SWEDEN

Blekinge Institute of Technology
Licentiate Dissertation Series No. 2011:05
ISSN 1650-2140
ISBN 978-91-7295-206-5

Published 2011
Printed by Kaserntryckeriet AB
Karlskrona 2011
Sweden

Abstract

Software has, for better or worse, become a core component in the structured management and manipulation of vast quantities of information, and is therefore central to many crucial services and infrastructures. However, hidden among the various benefits that the inclusion of software may bring is the potential of unwanted and unforeseen interactions, ranging from mere annoyances all the way up to full-blown catastrophes.

Overcoming adversities of this nature is a challenge shared with other engineering ventures, and there are many developed strategies that work towards eliminating various kinds of disturbances, assuming that it is possible to apply such strategies correctly. One approach in this regard, is to accept some anomalous behaviours as mere facts of life and make sure that the situations experienced are dealt with in an expeditious manner, while at the same time trying to discover, implement and improve safe-guards that can lessen adverse consequences in the event of future problems; in short, to embed resilience.

The work described in this thesis explores the foundations of software resilience, and thus covers the main resilience-enabling mechanisms, along with supporting tools, techniques and methods used to embed resilience. These instruments are dissected and analyzed from the perspective of stakeholders that have to operate on pre-existing, critical, large and heterogeneous subjects that are to some extent already up and running at the point of instrumentation. Finally, in the course of describing this subject, the thesis describes a demonstrator environment for self-healing activities in a partially damaged power grid, its construction details and the initial results of the study conducted in this environment.

Acknowledgements

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

... To my Supervisor, Rune Gustavsson; my Assistant Supervisor, Per Mellstrand; Jan Johansson and the people of the Pd MiB team at Sony-Ericsson; Raphael Caïre, Nouredine Hadjsaid, Seddik Bacha and the people of IDEA/G2Elab in Grenoble, France; colleagues and coworkers at Blekinge Institute of Technology; family, friends and those that I regrettably lost on the way ... *Thanks.*

Contents

List of Figures

1 Introduction

Bugs, flaws, defects, anomalies, mishaps, screw-ups, accidents, blunders, failures, blemishes, faults, slips, trips, crashes, glitches, shortcomings, imperfections, weaknesses, snafus, exploits, holes, failings, blisters, dents, cracks, marks, scratches, snags, malfunctions, oopses, disasters, drats, buggers, mess-ups, train wrecks, catastrophes and flukes.

The sheer number of words present in dictionaries and day-to-day conversations used for describing situations where the outcome of some particular event differs from expectations in some profound way is awe-inspiring. This extensive selection of synonyms, furthermore, seems to at least hint that either things go wrong more often than one would like or that we tend to be overly observant whenever things do go wrong. The most frightening thing is that we sometimes use these words to describe systems that, in spite of their failings, share an important role as parts of the general task of governing our lives. The reason why we accept this is probably because the benefits these systems bring seem to outweigh most of the kinks and quirks that they introduce.

In recent times, many of our critical or sensitive systems have been extended or enhanced with the inclusion of computing. Today, computing is not only an integral part of the critical systems themselves, it also acts as a sort of logical or structural glue *between* such systems, allowing them to interoperate in a more automated manner.

These operation critical systems – be it water and sewage management facilities or filing systems for medical records at hospitals – are riddled with challenges, some of which may be eased with the introduction of more refined computing. However, at the same time, computing introduces non-trivial unforeseen interactions into these systems.

Taking a step back to look at the larger picture, the ideal would, of course, be to somehow create systems that are without any snags or dents. 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 feedback loops which introduce non-linearities, such as reconfigurations (dynamic polymorphism), 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. In addition to this, system components are often executed in environments that did not, and to a certain extent could not be, 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[1] 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.

[1]Cascading is when the occurrence of a fault somewhere in a system propagates.

These concise principles, primarily derived from [37] 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;

- *Heterogeneity* – The components involved are of different origin in terms of both age, role and underlying technology, e.g. processors with different instruction sets, different fundamental architectures (*von neumann* versus *harvard* or *modified-harvard* for instance), or more technically, highly specialized Digital Signal Processors (DSPs) mixed with generic Central Processing Units (CPUs).
- *Concurrency* – Several dependent and independent computing tasks occur within roughly the same time-frame (within the borders of a system), all sharing or competing for the same limited resources.
- *Distributed* – The code, data, inputs and outputs of the different parts of the system are scattered across several devices, interlinked through some kind of communication bus or network.

To provide a more focused context, we direct our attention towards two major, critical and software-intensive classes of systems:

- *System Control and Data Acquisition (SCADA) for power-grid management*. In addition to their sensitive and critical nature, these systems are typically rich in legacy with components and technology stretching across a large time-span, having been modified and expanded upon to reflect and account for changes in the governed grid. By design, they are considerably brittle and, due in part to legacy reasons, vulnerable to an extremely large assortment of software and network security associated attack scenarios. Furthermore, it is likely that any specific SCADA solution is specialized to such a degree that its configuration and on-line presence must be treated as unique (single-instance, single-configuration). Thus, in contrast to more conventional software, it poses additional restrictions on the instrumentation and intervention that can be performed on the system once operational.
- *Mobile phones* represent a dramatic blurring of the conventional borders between embedded systems and more generic computers. They are typically comprised of literally hundreds of third-party hardware components and software libraries. Above all, they have a rather unique relationship with another legacy rich international infrastructure: the phone-network – with a large assortment of protocols and regulations that are challenging

both economically, politically and technically. Furthermore, through the transition to 'smart phones', mobile phones have gone from comparatively simple one-purpose embedded systems to multi-purpose devices that rival the size and complexity of modern desktop computers. Consequently, mobile phones are beginning to face similar obstacles when it comes to software security, piracy, development and maintenance.

Even though these two categories serve as the focal point and area of application for the research presented, the overall results and discussions are sufficiently generic to be applicable in many other computing endeavors. Briefly put, this thesis is about discovering the means and boundaries of resilience mechanisms with respect to software as part of software-intensive systems. The general focus of the thesis is thus how such mechanisms interact and can be taken advantage of to improve – or simply understand more about – the inner workings of these systems.

The structure of this thesis is as follows:

In chapter 2, *context*, a basic perspective is established on the terminology, ideas, history and problems that in part motivate the research but also serve as a context from which to consider the included articles. The chapter thus also acts as a primer intended for those with moderate insight into the specifics of computing and its associated technology.

In chapter 3, *structure*, the academic structure is detailed with respect to the research questions, the scientific approach and how these relate to subsequent chapters.

Chapters 4, 5 and 6 comprise the core of the contributions made. These concern (a) virtualization as a resilience mechanism and shows how it behaves in a wide range of applications, (b) the properties of the tools that can be used to explore and study such behavior and (c) how one can go about setting up experiment environments and experiments for studying the interaction between infrastructures.

Lastly, chapter 7 summarizes the work and validation of results and presents related work and future research possibilities along with lessons learned.

For sake of reference, the definitions to commonly used terms and abbreviations can be found in the *glossary* appendix section.

These chapters are written so they can be understood independently of each other, and may therefore – to the meticulous reader at least – appear as having slight redundancies or overlaps. However, if the contents of a certain chapter or section still appear confusing, undefined or even incomprehensible, return to chapter 2. If that does not help, blame the author.

2 Context

The goal 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 provide the context and background from which the other chapters can be readily 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.

Structurally speaking, the chapter is divided into two segments. The first one includes Sec. 2.1 and Sec. 2.3 and tries to answer the question *what is it that we are trying to make more resilient?* The second segment, beginning with Sec. 2.4, answers the question *which specific problems should we be more resilient against?*.

There are a few overarching points or arguments that are emphasized throughout the chapter:

- Software is not merely automated calculation.
- It is only through execution that software can present a behaviour.
- The descriptions used to articulate an envisioned software's intended behaviour do not necessarily depict the software's actual behaviour when executed.
- There is a large assortment of tools and dynamic processes involved when transforming a series of description into executing software. These tools and their respective configurations heavily influence the actual behaviour.
- The complexity of these tools may well supersede that of the system they are part of constructing.

In addition, the perspective in this chapter is mainly that of working from the point of existing, *developed* software rather than with the specific challenges of *developing* software, even though there usually is some overlap. The original software developers may be unavailable for different reasons. Thus, our approach towards exploring the mechanics behind software resilience relies heavily on our ability to experiment with a subject and understand its behaviours and transformations at a very low level even if this subject has not been explicitly prepared for this in advance. Some of the challenges involved are thus similar to the challenges related to debugging a mature target or to that of reverse-engineering.

2.1 Information Systems and Information Processing Systems

It is often claimed that software has, amongst other things, *behaviour*. Such claims have already been made several times in this chapter alone. Definition wise, behaviour 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 behaviour 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 behaviour.*

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 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).

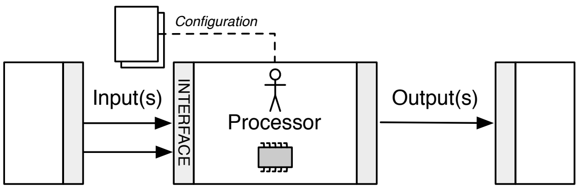


Figure 2.1: The abstract processor in an information processing system, where we only concern ourselves with the information streams (inputs, the interface, the processing and the outputs), not the specifics of the processor itself.

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 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 behaviour 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 *availability*. The merits of a system's online 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.

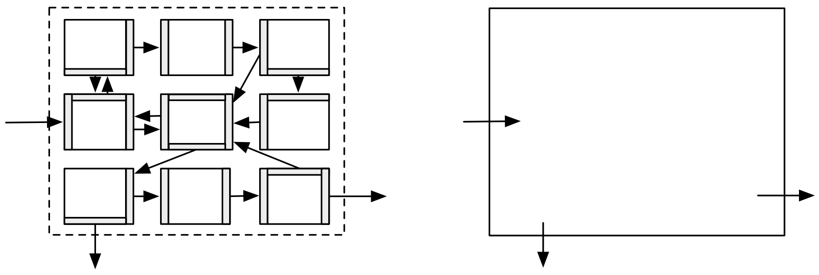


Figure 2.2: Systems of Systems, interconnected through interfaces. Viewed as a white-box (left) and as a black-box (right).

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 behaviour, let alone establish if the observed behaviour deviates from the intended or expected behaviour, or understand the chain of events that caused the observed behaviour?

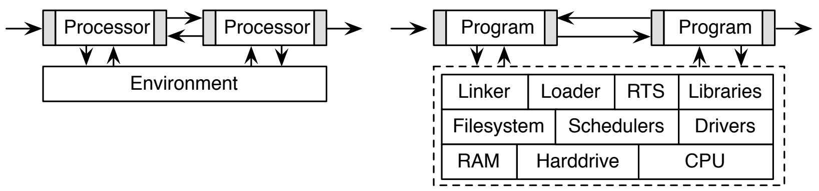


Figure 2.3: The enabling layer of an IPS, its environment.

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 loosing 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

[1]Something more than studying software simply for the pure joy of exploring the possibilities and structures of the abstract microcosm of computation.

flaws can be both difficult and time-consuming. Parts of this difficulty stems from 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 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 behaviour 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 Resilient Systems

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.

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) [1]. 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.

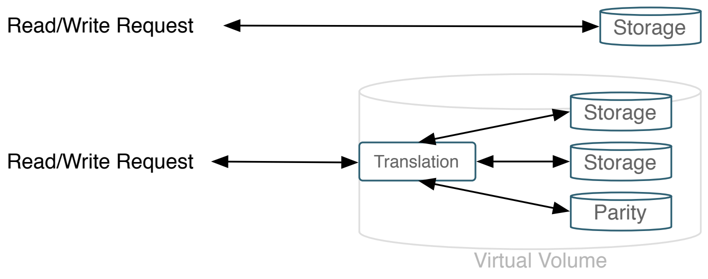


Figure 2.4: Illustration of direct access (top) compared to RAID-3 (bottom).

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.

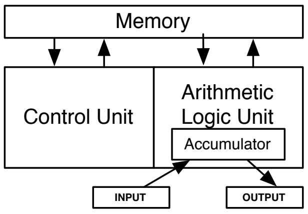


Figure 2.5: The Von Neumann architecture of a stored-program machine.

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 lookin 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.[2] 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 behaviour 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. 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[3].

[2]Indirectly, this means that the actual address is the result of a previously computation, using data from other memory locations or CPU registers.

[3]For a taste on how this used to be done, read up on a technique called *bank switching*. Although

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, *cascading data corruption*) into another one (*performance degradation*) while at the same time increasing overall complexity, by employing an absolutely central concept, *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.

2.3 Software and Software-Intensive Systems

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 *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 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,

somewhat outdated, the underlying principles are still relevant, with very interesting and far reaching consequences.

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.[4]

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.

- **Heterogeneous** because few components are ever likely to be similar and interchangeable. Lots of processors, varying from newer 64-bit Reduced In-

[4]For a slightly more academic touch on the subject, please refer to the *great principles of computing*, `http://greatprinciples.org`

struction 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 behaviour 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, retuned 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.

A primitive example of this is colloquially called *the dependency hell* problems that can be found at overlapping boundaries between subsystem such as shared libraries. Say that we have two programs, *prog1* and *prog2*, that rely heavily on a shared library, *shlib*. There is some kind of problem inside one of the functions exported in *shlib* that both *prog1* and *prog2* suffer from. The developers behind *prog1* is on a tight schedule and can neither wait for a new version of *shlib* to fix the issue, nor can they afford to develop their own replacement function. The compromise lies in writing a quick fix in *prog1* that works around the specifics of the bug in *shlib*. Meanwhile, the developers of *prog2* have responsibly contacted the vendor behind *shlib* with a report on the issue. Confident that the problems will be remedied shortly, the developers continue working on other parts of *prog2*. We can now fast forward a few weeks, months or years. Both programs are now out there in the wild, running on a variety of computers. Both *prog1* and *prog2* come bundled with the version of *shlib* that was used in development, both seemingly operating without problem. However, a subset of users installed *prog1* as soon as they got their hands on it, and then installed *prog2* later on. When doing so, the installer software complained that it found an older version of *shlib* and asked if it should be replaced with a newer, supposedly better, version of the same library. The local administrator agrees[5] and suddenly reports start pouring in that the business critical *prog1* has started to malfunction.

[5]or at least clicks **OK**, hoping that the annoying dialog box will disappear.

The issue is seldom as clear-cut and as in this example. Usually, the scale is considerably different, concerning not one or two libraries in conflict, but a large dependency graph of hundreds of libraries. Similarly, it is not one or two programs sharing a single resource but tens to hundreds of programs sharing many different kinds of resources. Obviously, the problem is not directly related to technology as such – even though technology can be used to mitigate the effects – but it is located somewhere in the intersection between management, politics and technology. It is for reasons such as these that the debacle around the Y2K[6] caused so much wide-spread panic. We have only had these kinds of systems for a very brief period of time and there are still unpleasant surprises waiting to spring[7].

2.3.1 SiS Example: SCADA

As an example of a software-intensive system rich in legacy, consider an instance of a 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.

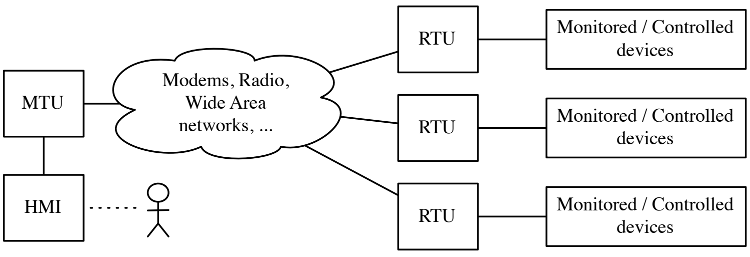

MTU
HMI
Modems, Radio, Wide Area networks, ...
RTU
RTU
RTU
Monitored / Controlled devices
Monitored / Controlled devices
Monitored / Controlled devices

Figure 2.6: A simplified SCADA Model

[6] A date representation problem in that many systems were representing years with only two digits of precision, something that worked fine in the span for the years between 1900 and 1999, but might prove problematic later on.

[7] An interesting collection of major such stories can be found at `http://www5.in.tum.de/~huckle/bugse.html`

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 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, and the common causes that precede them – to look at the *origin of software anomalies*[8].

[8]The term 'anomaly' is purposely used here instead of the more common and anthropomorphistic 'bug'. This is not to sound academic and pompous, but rather to emphasize the deviation from intended or expected behaviour and from actual or observed behaviour instead of some predestined and predetermined problem. In other words, this term allows ample room for interpretation. If, on the other hand, the behaviour has already been established as malign from the perspective of the

The principal claim on the supposed origin of software anomalies that is often discussed in software-engineering literature [38] 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, 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 [39].

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 [2] 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?

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

stakeholder in focus, the term bug will be used.

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 behaviour 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 behaviour 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.

2.4.1 Effects

To further the above discussion as to why various issues occur, we will proceed by categorizing the effects that are possible to measure, *observe*, during execution and which are independent of the programming language that was used during development. Thus, there is no speculation as to the link between an observed effect and some corresponding description or development process. Therefore, other commonly used or 'standard' classification schemes, such as [50] are not used here. Not all problems caused by software malfunction are the same when examined closely, and the various differences are reflected by the terminology used colloquially by developers and analysts; terms such as *'race condition'*, *'buffer overflow'*, *'memory leak'*, *'deadlock'*, *'stack overflow'* and a wide variety of other, more colorful names are often used to distinguish between different kinds of problems experienced. Some are very close to the underlying cause, others the effect and the rest is a mix of both.

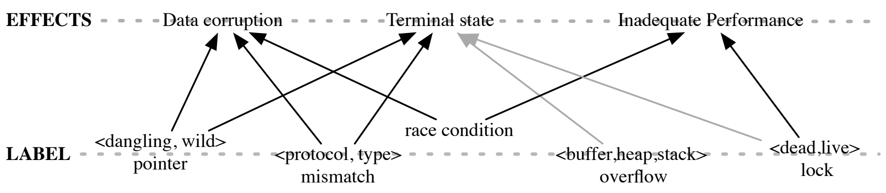


Figure 2.7: Two-tiered minimalistic taxonomy of observable effects and corresponding labels.

2.4.2 Tier 1 – Effects

The effects in this tier are not independent, as Fig. 2.7 illustrates. Often, there is a rapid succession of these effects that, *cascade* into each other at roughly the same time frame. This phenomenon causes the occurrence of one problem (distal cause) to further break other parts (proximal causes) that might previously have been behaving correctly and that were perhaps even verified as functioning according to specification. However, thanks to the huge amount of instructions processed in a fragment of a second by modern computers and the comparably slow response time of a human observer, the problem that was detected is likely to be the proximal one rather than the initial.

Data Corruption

Data corruption means that instead of what was supposed to be written to a particular place in memory, to some secondary storage or conveyed to some distant device over a network connection, something else, likely garbled, is written. It might also be the case that the data itself are correct but are instead saved to an erroneous location/address, and if that location is in use by something else, it becomes an additional occurrence of data corruption. The way this manifests depends on what the data is used for and if the data exceeds some other virtual unenforced boundary and flows into adjacent memory used for other purposes. This may vary considerably depending on where the corruption takes place; a rule of thumb is that the closer to the processing device data get corrupted, the sooner immediate effects can be observed. For example, primary memory and especially memory used as stack is considerably more sensitive than both secondary storage or remote connections, due to the amount and frequency of writes taking place and the scope of safe guards in place.

Terminal State

In contrast to the more generic use of *state* for describing 'the way things are at a certain moment in time', a terminal state is reached when the execution of a program has, for any reason, come to a complete stop. This does not necessarily have to be a bad thing, but it depends, of course, on the context and the timing of the event that initiated the termination. A program terminating as part of its programming, or a program being terminated because there is nothing left to do, may not be a big deal. However, a program suddenly terminating during a critical transaction is a completely different affair. There are quite a few parts involved in the operation of a computer that can induce a terminal state, acting as safeguards against other failures or more nefarious activities. Safeguards are often found in the shape of exception and trap handlers or in the form of memory access protection such as guard pages, read-only memory addresses, and many more.

Inadequate Performance

Performance is a tricky property for many reasons, and there are several central terms thats need to be considered. Starting with the unit of measurement, we first need to find out if we are interested in performance in relation to the amount of processing cycles that some piece of code requires, or if the problem is related to an external clock measuring how many milliseconds that have elapsed? Then comes the precision needed. Should the measurements be exact, or can they be fine-grained or perhaps even coarse-grained? Each of these terms carry additional problems. One is that the demarcation between adequate and inadequate performance is often blurred. The notable exception is the case when a terminal state has not been reached and progression has halted (look at the tier2 labels live and deadlock). To exemplify this, one can look at time constrained settings where there are strict deadlines for various activities, which is the case in a variety of critical areas such as medical appliances and vehicle control systems.

A more interesting, follow-up example are graphic engines. Among their various challenges lies the fact that there is a certain real-time component based on the properties of the display device and of the user. The refresh rate of the display sets an upper threshold of how many image frames that can be drawn, and the perception of the user sets the lower threshold (somewhere around 12-25fps). At the same time, graphic engines are operating on a best effort basis, meaning that in a sense, there is always a need for more resources, and the main task is using the available ones as efficiently as possible. Given the heterogeneity of underlying hardware configurations, along with concurrent tasks utilizing resources differently in unpredictable ways, good engines are constantly adapting to the current situation, switching algorithms and structures based on the current choke-point [40]. Thus, there are certain contexts, of which real-time graphics engines are a good example, where sufficient or adequate performance in respect to the quality of service provided is a very dynamic target.

2.4.3 Tier 2 – Label

Based on the observable events from the first tier, we now connect these to common labels used to reference a certain event in relation to a known structural flaw or malign chain of events. For each label, there are a few properties well worth considering, such as:

- Observer effect sensitivity – describes how the mere act of observing (gathering measurements) influences the behaviour of the measured system.
- Repeatability – called *reproducibility* by some – describes how difficult it is to exactly mimic the circumstances which triggered the effect in the first place.
- Severity –referring to possible and likely consequences.

Race Condition

Race conditions cover all kinds of situations where we have two or more concurrent tasks with some dependency on a shared resource. The problem appears when one task alters the shared resource in a way not accounted for by the others. Since timing in regard to the shared resource is a large contributor to when and how the effect is triggered and what consequences will follow, race conditions are sensitive to everything that may affect such timing. This can make them hard to catch in the act as both occurrence and impact are difficult to determine.

Deadlock and Livelock

If we would catalog all labels further, both deadlock and livelock would fall on the same page as race conditions, and they have similar properties when it comes to repeatability. Deadlocks are, however, particularly picky in regard to which concurrency-prerequisites that have to be fulfilled for them to occur[9] [3], Fig. 2.8, whereas livelock also requires repeated testing (without any other computation being performed) for something which is supposed to happen but never does, like checking the shared resource for some change or value that is supposed to be set by the other task but never is. In terms of detection, deadlocks are a lot easier to discover than other kinds of concurrency problems, since the effect is immediate (several processes stop proceeding) and the impact is typically critical, with algorithmic approaches for automatic detection available. Livelocks have a similar impact but are a bit trickier to detect because execution continues; the instructions performed simply do not lead to progress in computation.

[9]Mutual Exclusion, Hold and Wait, No Preemption and Circular Dependency

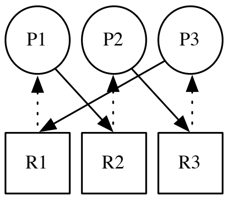


Figure 2.8: An illustration of the deadlock condition.

Buffer, Heap and Stack Overflow

All these kinds of overflow problems describe some boundary being unenforced in the implementation of a data structure or type which, instead of returning an error, leads to an overwrite of some other data that happen to be stored in a location nearby. This fits the definition of data corruption effects pretty well.

When a buffer overflow occurs, the data structure, which is some sort of array (memory region), is being indexed or addressed outside of its allocated space. Heap and stack overflows, on the other hand, describe not the structure being targeted but rather where the corruption takes place. Buffer overflows are reasonably easy to find and deal with when both the targeted structure and the supposed boundary are known. Unfortunately, buffer overflows occur quite frequently in software partly developed in languages that expose a lot of detail. If the subsequent data corruption hits user data or system state variables rather than crash-prone areas, chances are they will lead to more complicated problems.

Dangling, Wild Pointer

Dangling or wild- pointers are references to memory locations or objects that for some reason are no longer valid in the sense that they either reference the wrong (although this may be hard to determine) object or address. These situations arise when pointers are allowed to be arithmetically changed when using them without proper initialization or when state-variables containing address information are not updated after referenced resources have been (de)allocated or moved.

Protocol / Type Mismatch

As mentioned in the beginning of Sect. 2.4, there are a multitude of protocols that developers are supposed to follow in a variety of degrees. These protocols depend on an interface through which data are passed in accordance with a set of rules. These rules dictate the formatting and sequence of such exchange (a protocol).

Information in virtually all current machines is represented in binary at the lower levels, a representation that there are several ways of interpreting with some being compatible with others. This allows for the establishment of *type*, a very common construct in programming languages at all levels. Interfaces and, by association, protocols are typically written with this distinction of type, in mind. Some languages allow for the direct and implicit (automatic) conversion between types, and languages with extensive support for the Object Oriented Programming paradigm are exceptionally rich in type-abstractions and even type-hierarchies.

When a protocol is implemented in a language that allows for conversion between types or several types to pass through an interface, chances are that the implementation may accept types that have not been considered for that particular use, risking potential consequences such as undesired state transitions and data corruption.

Resource Leak

A large portion of current programming concerns resource management. Programs allocate resources from some shared pool facilitated by some kind of managerial subsystem, most commonly the operating system kernel. Resources can be anything from primary memory (Random Access Memory (RAM)) to more abstract ones such as handles for threads, interprocess communication and Transmission Control Protocol (TCP) ports amongst several others.

Typically, a program requests an allocation regarding a desired resource some time in advance of its intended usage. When a previously allocated resource is no longer needed, most resource management schemes require that the program performs a deallocation as a hand-off to indicate that said resource can freely be allocated by other processes.

It follows then that if a program fails to indicate that a resource is no longer needed, the allocation might last for either the lifespan of the process or (depending on resource management technique) worse – the lifespan of the system itself. While this may seem like a minor detail, there are a few factors in play that may escalate further consequences considerably:

- Allocation rate – how often leaky resource allocation requests are invoked.
- Allocation size – how large leaky resource allocation requests are invoked in respect to the total amount of that resource available.
- Process lifespan – various system services and server programs tend to live longer (in some cases, indefinitely) than regular tasks. The lifespan of a process is interesting when termination would lead to deallocation of all

associated resources as a termination or forced termination might then free up resources for other tasks to use.

All these factors affect the time it takes to reach a state of *resource starvation*, the point where the pool of allocatable resources is drained, meaning that any new allocation requests fail. For many programs, the result of a failed allocation is detrimental, forcing execution to take often untested and unconsidered paths with cascading effects covering the entire span of possible outcomes.

Remarks

There are cases that do not fall neatly into these categories. A *double free*, for instance, is a special case with C-malloc style memory allocator functions where deallocation is performed twice on the same address and manifests itself in three different ways: as a crash, as data corruption of program specific data or as data corruption of program control structure data.

- In order to manage the allocation and deallocation of memory, some control structures are needed in order to keep track of allocations that have been made. If a block of heap memory is deallocated twice without any other actions taken in between deallocation requests, the second deallocation will have an argument that points to an address not referenced by these structures. This situation may be detected by the memory manager and will yield a crash or an abort.
- If, however, an allocation is made between the first and the second deallocation call, chances are that the deallocation will affect memory currently in use by some other part of the program and which will subsequently be used as a result to other allocation requests. This leads to a non-intuitive corner case of race conditions where the shared resource is unintentional, which in turn is very likely to result in data corruption.
- Finally, some memory managers make little or no effort to try to secure the data in their own control structures. The second deallocation request may then, instead of a crash or exception, lead to overwrites of data in such control structures, e.g. next and previous pointers in double-linked lists.

One large contributor to situations like double-free is called *aliasing*. Aliasing occurs frequently when dealing with pointers but also exists in systems that use a more restrictive form of pointers, *references*. Aliasing means that an allocated block of memory is referenced by several variables at the same time, either at the base or at an offset. This has an effect on performance in that it restricts the amount of optimizations that a compiler can perform safely and it is particularly dangerous when aliasing occurs between functions because the programmer would then have to take into account how all functions manage memory.

Albeit a subject much too involved to bring up here, it is worth mentioning that all these labels also have a software security dimension to them. To name a few:

- Deadlocks and Livelocks are sought after as targets for denial-of-service types of attacks, with the primary goal being to overload a victim device to the point where it cannot process legitimate requests, or as the intermediate goal when trying to expose other attackable surfaces.
- Memory mismanagement in the sense of heap or stack buffer overflows has for long been the primary source of software security commotion. Despite many clever methods of protection, exploitation techniques have for the most part been able to keep up. Advanced examples currently include Return- Oriented Programming (ROP) and heap-spraying.
- While it is impossible to establish control over every dangling or wild pointer (in fact, the address of the pointer is not necessarily controllable) there are certain corner cases where they can be exploited. For instance, when the address can be controlled or determined, even if just partially, and it can then be used as an array with user-supplied offset.
- Protocol and type- mismatch scenarios are broader and are thus likely to branch out in any of the above. However, they also suffer from application-specific attacks of their own, particularly in domains with complicated or dynamic type models and ambiguous protocols. Examples of this are most things web-related, cases such as cross-site scripting, cross-site request forgery, Structured Query Language (SQL) injections and many more.

The main reason why software security is relevant to the discussion in this thesis is not simply the fact that it can be used to illustrate how severe the consequences can be from seemingly harmless mistakes, but more importantly that the area is highly developed when it comes to taking advantage of the dynamic playing field, and that there are valuable lessons to be learned from such efforts. There is also some additional overlap that extends towards conflict in the sense that the mechanisms that can be used to measure, monitor and experiment with a subject also open up for more nefarious activities that are the distinct target of certain security efforts.

3 Structure

The layout of this chapter is as follows:

In the first section, *Approach*, the fundamental issues that we work towards resolving are presented in the context of a *mission statement*. This mission statement also covers academic and industrial efforts that, to some degree, act as influence or bias to the direction and underlying assumptions of the material presented. From this position, a series of research questions are addressed.

The second and final section, *Contributions*, introduces the papers included in relation to the research questions presented, their methodological aspects and the publication forms. This section also describes how these papers interrelate and which kinds of alterations that have been made in order for them to better fit this thesis.

3.1 Approach

It is worthwhile to note that the work presented herein is in no way conclusive nor are its questions to be considered fully answered. That is not to say that this body of work is not a relevant and structurally sound step on the way. The initial goal is thus to establish explorative models, methods, tools and experimental environments that may serve future endeavors.

3.1.1 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. This ICT requires a reliable power 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 [59, 60]. In addition to this, indirect threats have been partially responsible for large incidents, such as the case with the BLASTER worm's contribution to the much discussed northeastern blackout of 2003 in the USA [61].

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, 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 [51], 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 is to fully investigate the ways in which all the peculiar details of software can support or undermine infrastructure resilience.

3.1.2 Research Questions

Working from the perspective of the mission statement, the first research question established was that of *mechanisms*. This entails working from the assumption that it is indeed possible to establish different degrees of resilience in software-intensive systems that are used as information processing systems in critical infrastructures. Tied to this, is the hypothesis that the resilience of these systems share a set of controlling principles. From this standpoint, we have found several recurring patterns connected to effects that were beneficial to some degree for the resilience of the system, while at the same time having notable consequences that could be perceived as harmful. It is from this realization that the second research question, concerning *caveats*, stems. This question addresses to which extent such drawbacks can be managed. At this point, it was clear that in order to properly answer this question, certain experiments needed to be performed. During the design phase of these experiments, we saw that the models and technical facilities at our disposal were inadequate in several respects. Before we could begin resolving the problem, this situation had to be remedied. It is here that the third and fourth research questions, concerning *support* and experiment *environment*, are posed.

The research questions are detailed in the following manner:

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

Software can (during execution) present, at least, two different degrees of resilience; *fully resilient* in regards to a specific disturbance, or *non-resilient*. These degrees depend in part on the structured control of the 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 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 morale 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

[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.

to critical software-intensive infrastructures, i.e. how they support monitoring, measuring, altering, 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, and which guidelines should regulate such experimentation?***

Computing has long had the oddity that the arguably best experiment environment for conducting experiments is unsurprisingly enough, 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 behaviours 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 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 the power-grid) there are suddenly several additional complexities brought on by foreign technologies combined with a socio-political dimension. At such a stage, the composition and services of the experimental environment itself becomes a legitimate subject of study.

The results that stem from the study of these questions and discussion relevant to the validity of those results can be found in Chapter 7, *Conclusions*.

3.2 Contributions

The main contributions presented are as follows:

3.2.1 Paper I – The use and misuse of Virtualization

In this paper 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 current examples, along with 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 paper has a strong focus on the interplay between philosophy, methodology and technology.

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 paper, 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 paper has been submitted for publication in ACM Transactions on Computer Systems, review pending.

3.2.2 Paper II- Retooling Systemic Software Debugging

This paper is primarily about the examination of key tools used when debugging software. This examination is done in order to determine which fundamental dependencies such tools have and which incompatibilities that may stem from those dependencies. The consequences of likely incompatibilities are then illustrated by an industrial case that involves the transition from a closed embedded setting to a more open one and the nature of the challenges that arise in maintaining the dependencies of the tools involved. Combining these experiences, the paper concludes with preliminary steps towards the advancement of coordination between such tools. Thus, this paper has an industrial/technical focus and concerns problems that will likely play an important role in the near future.

As per the abstract; *There are a few major ancillary tools that have long supported the frustrating and time-consuming part of programming that is debugging. These tools are, in no particular order, the symbolic debugger, the profiler, the tracer and finally the crash dump analyzer. While the effort to keep these tools in-line with the overall progress of development tools and developed systems has been strong thus far, there are troubling signs of phenomena ahead which may seriously hinder further advancement. One such sign is the role of the developer shifting from being in charge and control of the development of one distinct piece of software to, instead, combining a large assortment of third party components and libraries into a common service or platform. To help such a situation, the underlying problems and how the properties of the tools involved interact are first examined. Finally, this shift is exemplified through an industrial case and later approached by outlining remedial complements.*

The paper has been submitted to the International Workshop on Program Debugging at the 35th IEEE COMPSAC Conference, review pending.

3.2.3 Paper III- Experimenting with Infrastructures

In this paper, we describe 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 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 behaviours 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.*

The paper was presented at the fifth international CRIS conference on Critical Infrastructures (CRIS2010) and is published in the *IEEE X-plore* [4].

3.2.4 Comments

It should be noted that the related publications (Self-healing and Resilient Critical Infrastructures [5], Analyzing Infrastructure Malfunction [6]) and (The empowered user - The critical interface to critical infrastructures [7]) have been omitted due to differences in scope but also because the relevance of the contents in regards to this thesis is superseded by the included publications.

It should also be noted that these publications deviate slightly from their published or submitted forms as formatting and layout have all been altered to fit the overall format of the thesis. This includes references being merged and moved to the references chapter. In addition, some explanations have been widened or clarified and any more substantial alterations in respect to argumentation, definitions, results and figures[2] are listed in the *errata* section at the end of each chapter.

[2]The figures that had a rasterized source-format in respective publications have been vectorized for the sake of readability.

4 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 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.

4.1 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 and so on.

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 [6].

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 behaviours of a specific software system, explicitly hides these features and can only be considered a primitive model for the executing software, at best [8].

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.

[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.

4.2 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 behaviours) the sort of complex software-intensive systems that were depicted in the previous section.

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, namely running several operating systems (guests) inside another (host), Fig. 4.1. This particular form of application will not be covered in detail here, but such discussion can be readily found elsewhere [9, 10].

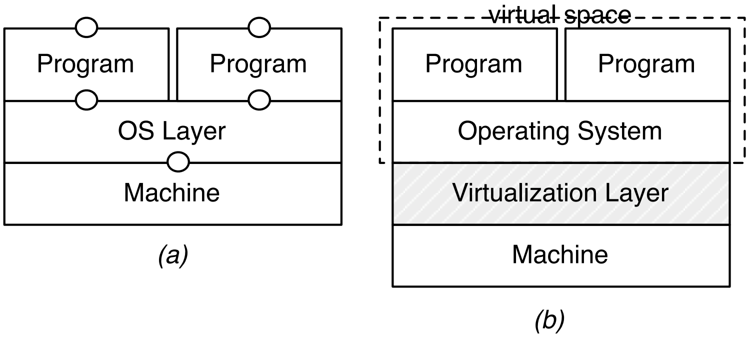


Figure 4.1: Crude model of a program running on a multitasking OS with process separation *(a)* alongside a hypervisor / VMM model *(b)*.

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 here modeled as

$$computing = code + execution \tag{4.1}$$

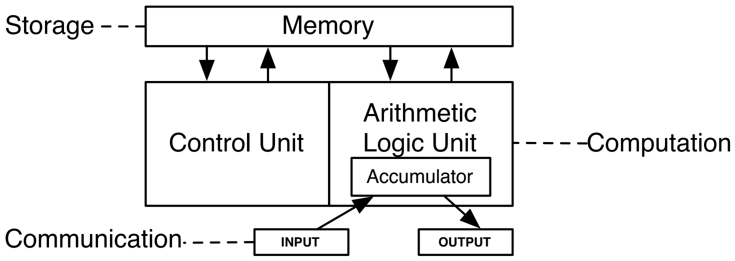


Figure 4.2: Von Neumann architecture and its virtualizable components.

Using the von Neumann architecture, as per (Fig. 4.2), 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.

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[2], 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

[2]An instruction set where most or all of the possible binary sequences have corresponding instructions.

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 [62].

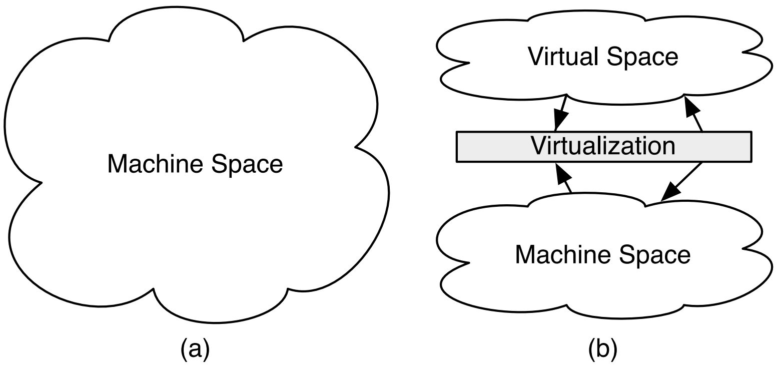


Figure 4.3: A virtualization, in effect, slices a state space into two parts.

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.

[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.

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.

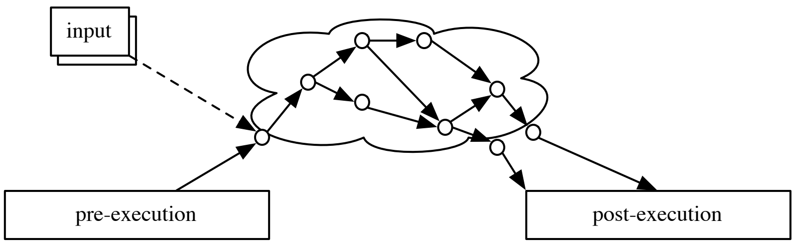


Figure 4.4: The virtualization ideal.

In essence, what a virtualization aims to establish, as illustrated by (Fig. 4.3), 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. 4.4) 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 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 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.

4.3 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[5]. 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 [11].

[4]Or perhaps even more distinct, when calling a function through a pointer.

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

Compatibility[6] 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, Rune Eveywhere (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) [63] alongside the Wine Is Not an Emulator (WINE) [64] 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 [12]. 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 systemic optimization strategies[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 [13].

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

[6]Also referred to in some settings as *mobility* or *portability* even though these terms are not entirely synonymous.

[7]This is more fine-grained than, for instance, the compilation directives of optimizing either for speed or for size.

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 [1]. 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 debugging. 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 *lowering* 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.

4.4 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*.

4.4.1 External and Persistent States

Expanding on the abstract figure of the virtualization ideal (Fig. 4.4), we get (Fig. 4.5) where the two major additions are the notions of *persistent* states and of *external* states[8]. 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 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

[8]Not all external states are persistent but all persistent states are by definition external.

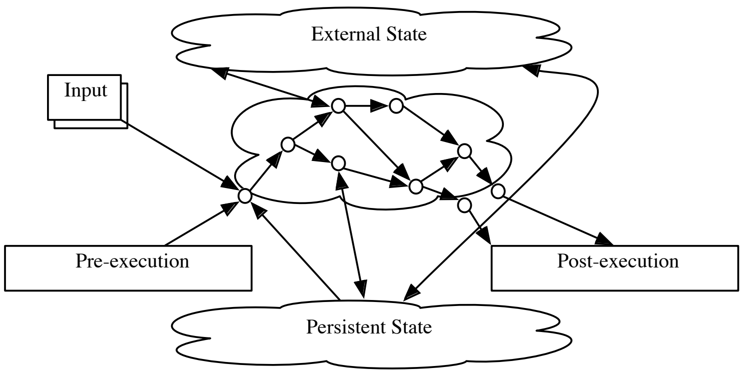


Figure 4.5: The fundamental problem.

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 *one to one* or *one to many*. This also relates to the performance model (Eq. 4.2, Sect. 4.4.3), 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 final code[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

[9]Zero matching instructions when the statement has been removed through subexpression elimination or similar form of optimization.

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 contra-productive to several virtualization goals. As covered from a security context in [14] 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.

4.4.2 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.

```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 (*)())dlsym(handle, "infun");
    if (fp) fp();
  }

  return 0;
}
```

Figure 4.6: 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.

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. 4.6) in relation to (Fig. 4.7) 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. 4.6), 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_PRELOAD environment variable used by, amongst others, the GNU LD linker [65]. In the (Fig. 4.7) case, things go even further. Not only does the (Fig. 4.6) still apply, we also need to know the contents and exact order of what was being passed through the TCP socket. As the word implies, the *dynamic* case requires one to consider a time-frame as well; 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[10] [41]. 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 4.7: 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.

4.4.3 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.

[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.

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 CPU frequency, as suggested in [42]. 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. 4.2.

$$overhead = invocation\ frequency \times translation\ cost \tag{4.2}$$

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. 4.4.1 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' [43] 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[11]. Another way to illustrate the problem would be in terms of interpreter design and by balancing which parts of the built-in libraries of a 'scripting' programming language, such as Ruby or Python, that are implemented in the code of the virtual space and which parts that are implemented in the code of the machine. Parts that are likely to have either a high invocation frequency, a large translation overhead, or both, get pushed to a lower level, which essentially "solves" performance problems with virtualization by either stripping away said virtualization and/or by increasing the complexity of the machine with further consequences down the line.

4.4.4 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.

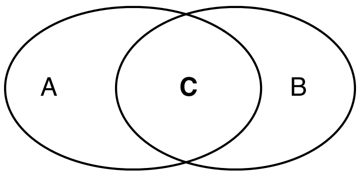


Figure 4.8: The feature set of two machines (a and b), and the feature-set (c) of an abstract machine that a virtualization implements.

[11]These contents can also act as external state-holders as per the model in (Fig. 4.5).

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. 4.8 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 [52]) 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.

4.4.5 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 behaviours 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 [15]. 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 behaviours, 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 [16] 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 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. 4.4.3. 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 [66] 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.

4.4.6 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 ROP [17]-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 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 [67] 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 [68]. Sometimes, the virtualization-friendly features of hardware are taken control of in order to virtualize the host Operating System (OS) itself [69]. 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 case with many copy-protection systems and other forms of surreptitious software [39] 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 [18] 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 behaviours 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. 4.12 for an example).

4.5 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 to 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.

4.5.1 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. 4.9) can be evaluated. We have previously worked on such environments for the evaluation of infrastructure protection measures [19] 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 [44]. The starting point is an initial view of the system, (Fig. 4.10) which refers to the analyst's current

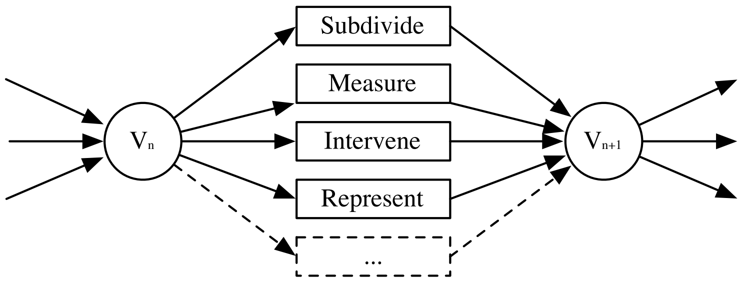


Figure 4.9: At each step, we have general activities that refine our view of the system or physically change it.

perspective of the system. From here, one out of several generic actions can be carried out, (Fig.4.9). 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.
- 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.4.11).
- 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. 4.1, 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

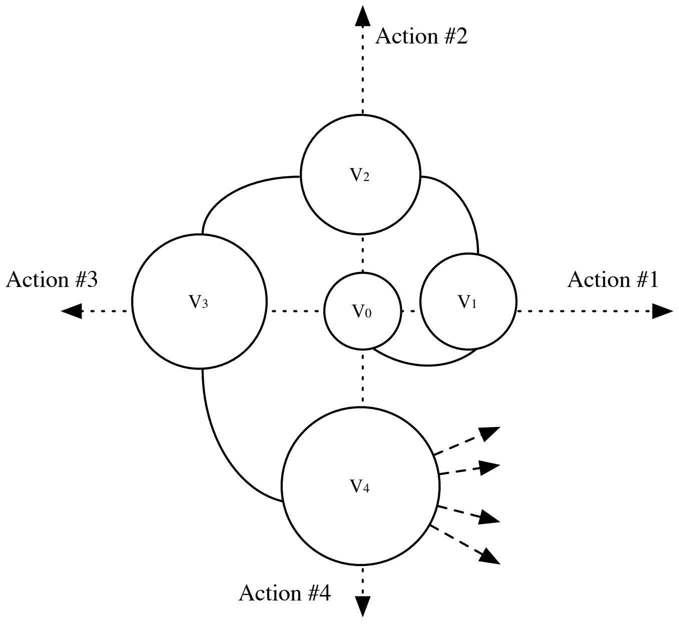


Figure 4.10: An interactive, explorative process. Starting from an initial view of the system (v0), we progressively achieve a refined understanding of its constituents and their respective behaviours.

to link with similar or related environments such as the internet. Preliminary work to that effect can be found in [4].

4.5.2 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. 4.12) 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

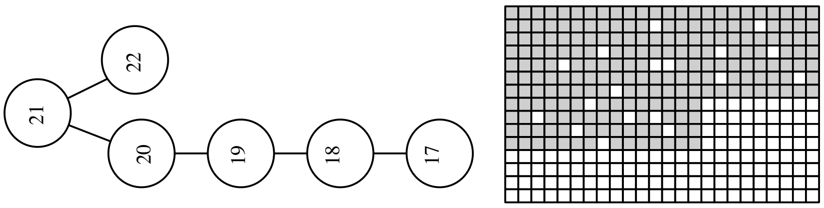


Figure 4.11: Two examples of non-native representations used to hi-light specific attributes. To the left, there is a poorly balanced binary tree and to the right, some mild fragmentation in a memory or file system.

or processing like encryption or compression, dilute the distinction between primary resources[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.

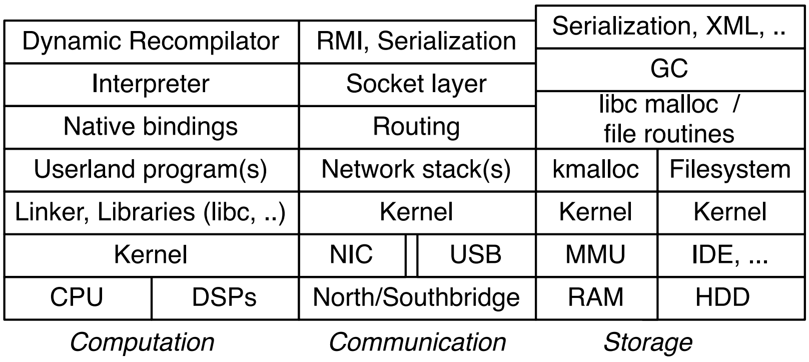


Figure 4.12: Illustration of some of the abstractions typically involved in making a program operate.

To begin applying this principle, start with an overhead perspective of resources and virtualizations involved in the targeted system, along with interdependen-

[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.

cies. 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) [20].

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.

4.5.3 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.4.13) as a 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"* [53], 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 lenience towards non-conforming implementations or outright ambiguous specifications are quite severe.

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

```
#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 4.13: A small snippet from a text conversion routine.

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 [21] [22]. 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.

4.5.4 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[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. 4.1, 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. 4.4 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:

[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.

- *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.

- *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 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* [23]. 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.

4.5.5 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 behaviours 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. 4.3) 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 Retooling Software Debugging

The layout of this chapter is as follows:

Context covers the perspective and background that served as motivation for this study. *Toolsuite* then itemizes the composition and mechanics of the major categories of tools typically used when debugging. Following that, the *SiS (Software intensive Systems) transition* exemplifies the challenges involved, illustrates the process of restructuring a debugging toolbox and discusses some important barriers experienced when doing so. Lastly, the section *Moving Forward* covers a few of the major recent developments in terms of debugging tools, the challenges involved when using these tools and suggests how these can be modified or complemented to improve the state of the art.

5.1 Context

Historically speaking, the tools for debugging used in any engineering endeavor have appeared out of an interesting mixture of ingenuity and immediate need. It is something of a good engineering trait to be able to improvise and adapt, or create, tools able to deal with a situation where something did not go quite as planned. Unsurprisingly, development and debugging have to some extent co-evolved so that neither seems to really get ahead of the other. In this respect, software is a bit odd in that the same sort of techniques and technology that are used to craft software are also used to create debugging tools; software is used as aid for the study of software.

Since software development is a systematic rather than a stochastic process, there is also a strong methodological aspect present. Part of this aspect is that some methods still in use are implicit and not yet formalized. Thus, a lot of work has been done in terms of distilling implicit practices into formalized methods, particularly in terms of developing, verifying and validating software in view of it being considered as a product in itself. In this regard, the troubleshooting aspect has tacitly been ignored in favor of efforts directed at improving testing, with the perspective that the two are somehow related. This is an understandable position if you agree with the notion that when a formal development process is completely successful, the end-systems will be functional so that debugging becomes, at worst, a minor inconvenience.

When reviewing the suggestions on methods for debugging software directed towards engineers [38, 45, 46], it should be noted that they range between playful mind games such as puzzle solving and role-playing, all the way to hypothetic-deductive reasoning. The common denominator is that they all rely on a hypothesis, that is split between either a vague or intuitive guess or hunch, and a hierarchical, binary quantifiable notion.

An example as to the latter can be found in [24], but this example also emphasizes the problem in establishing formal debugging methods: to quote; *"Unlike testing, debugging concerns not only the program specification and source code, but also – and more essentially – the various causes of errors. The major subject of debugging is causal reasoning within an integrated process of developing, selecting, verifying and modifying hypothesizes about errors. Therefore, to establish a formal framework we need not only theories on specification, semantics and behavior, but also theories on error analysis and causal reasoning."*. However, reworking theoretical philosophy treatments on causal reasoning [47, 48] into an applied debugging perspective, will probably be an arduous task.

Our experiences in regards to the form of hypothesizes useful for debugging, lean towards the more explorative and experiment-oriented. A suggested reason for this, and thus an unverified hypothesis in itself, is based on *when* in the development stages of a software that the developer needs to debug and that there is some tipping point in favor of one over the other.

To elaborate: on the one side, there are developers actively working on some future software-based product or service. These developers are strictly adhering to some preset development method where there are many short cycles between writing a portion of code, having it tested, observing that a test fails and adjusting the corresponding portion of code until the test finally succeeds. On the other side, there are developers working on tuning or otherwise improving some software solution that is to some extent already up and running or even deployed. These developers receive reports from various stakeholders on experienced problems; reports that have been filtered through several layers of customer support. In such a scenario, the software solution can be perceived as aging, meaning that it has not been updated or changed to reflect other external changes to components that the software either indirectly (operating system, hardware configuration) or directly (shared libraries) depends on.

Even though the problems experienced in early development may, of course, still occur in the later, more mature stages – assuming that the testing and verification efforts have not been sufficiently rigorous and thorough – there is still a large series of environmental factors in play. Such factors include be variations in the executing hardware (partial damages from overheating or other physical disturbances), cascade- effects from other third-party software that modifies shared resources in unexpected ways, and even undesired interventions (cracking, viruses, rootkits, etc.). All these factors can affect the end behavior that is reported and thus needs to be taken into account when investigating reports and crashes.

Thus, it is not unreasonable to think that developers who primarily work with software in the earlier phases drift towards a more strict view of hypothesizes in regards to debugging, particularly if they possess a very intimate understanding of how things fit together. Meanwhile, those involved in the latter stages may instead drift towards a more intuitive perspective. Subsequently, the latter category should also be more prone to rely on an assortment of smaller experiments to extract relevant feedback from the stakeholder input.

The argumentation used in the remainder of this chapter is based on the latter view.

5.2 Toolsuite

In the previous section, it was assumed that the interplay and connection between tools, method and the problem at hand is central, and that our approach to debugging is *systemic*, meaning that it focuses on a broader range of interaction between large systems or components, rather than tied down to a smaller program or algorithm implementation. With this assumption as the starting point, there is sufficient reason to make a quick re-evaluation of the tools (or rather, categories of tools) that are likely to be part of the debugging toolbox (which, to a large degree, can be distilled from the features present in modern integrated development environments).

5.2.1 Symbolic Debugger

The Symbolic or source-level debugging (usually referred to as simply 'the debugger') is a term that essentially describes the way an analyst[1] is interacting with the tool. Source-level or symbolic means that it is the source-code that the gathered measurements and specified interventions are referenced through. The key features involved are that of data control and execution-flow control, both dealing with *what is to be measured or manipulated* and *at what point (conditions) during execution* that these actions should occur. Both interventions and measurements are regulated using *breakpoints*.

Breakpoints can, from an abstract perspective, be *any controllable interrupt* (or trap) during code execution that the debugger is able to introduce (and remove). When an interrupt is triggered, the control is diverted to a handler routine inside the debugger. This may involve support of the underlying machine or environment through interfaces such as ptrace or JTAG, but can also be done

[1]As an effort to avoid language such as 'the debugger debugged the bug using a debugger', the term 'analyst' is used here to refer to some person investigating an issue, and 'anomaly' rather than 'bug' to emphasize the deviation between intended or expected behavior and the behavior experienced or measured, that such a deviation is not necessarily malign, and that it may well be subject for interpretation and discussion.

in a more direct fashion by simply manipulating the stored program, *target*, in memory. 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*.

This is, undoubtedly, a very brief and shallow treatment of the symbolic debugger as a principal debugging aid. Yet, there are few alternatives to examining the source code of open debuggers such as GDB [70], perhaps with [71] and [49] as references or starting points. However, the above is sufficient background to enumerate some of the key issues in view of the upcoming discussion in Sec. 5.3 and Sec. 5.4:

- There is a large disparity 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*.
- The presence and ability of a symbolic debugger can be both detected, circumvented (anti-debugging [25]), misdirected ([26]) and is further hindered by common security measures such as ASLR [27], DEP/WX [66], etc.
- Many low-level dynamic states needs to be tracked in order to retain truthfulness (relocations, trampolines, variable-instruction length decoding, state sensitive instruction sets, etc.), which is an important trait of any debugging tool [28].
- The debugger kernel has to both track, and be aware, of threading schedulers and other concurrency techniques.
- The source-code level of representation requires a special build that retains private symbols and, preferably, excludes most or all optimizations.

5.2.2 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[2].

[2]Finding anomalies in input/output or the lack of output corresponding to some input, is how a person would note that something is wrong in the first place, rather than through the trace evidence that can be used to explain why the observed anomaly occurred.

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:

- 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* (timestamp) the sample was gathered.
- 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.
- There is no clear or default reference model (e.g. source-code) to appropriately compare and study.
- 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.

5.2.3 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*.

[3]Even though it can be integrated as a part of a specialized build of the software.]

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 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 this 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):

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

5.2.4 Crash-Dump Analyzer

The last tool category to be covered in this chapter 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:

- 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.
- 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.
- 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.
- 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.
- All data present were probably not updated in the same instant. Thus, there is ample room to extract temporal, and at times causal, links between data (apart from generating call traces).

5.3 SiS Transition

Software has undoubtedly gone through a series of daunting changes throughout the years, and there is no apparent end in sight. Among all these smaller shifts, some have produced notable consequences for debugging and need to be emphasized, as they concern interpreters, dynamic linking and dynamic recompilation. The common denominator is the keyword *dynamic* in the sense that key details of the current composition of a program is only ever known and accessible during execution.

The direct consequence of the dynamic side to computing is that it dissolves the notion of software as bounded computations with a strict view of program input as being only data to process. Furthermore, computations can be gradually specialized to suit the execution and input patterns unique to one specific instance of a software. In this way, the software makes the best use of current run-time conditions, which subsequently has led to programs that are highly dynamic.

In addition, the computing hardware is to a large extent more heterogeneous than a couple of years back. Partially through the main-stream transition to hybrid 32/64- bit processors, offloading more generic computing tasks to a Graphics Processing Unit (GPU) and other specialized DSPs and because of additional CPU architectures introduced to balance computing power with energy efficiency, e.g. x86/ARM hybrids, it is now reasonable to assume that some vendors and services will rely on code being dynamically translated from one architecture to the other and back, depending on current running conditions. In fact, this has already occurred to some extent (although not transparently in the same execution run) in Apple's transition from PowerPC to X86 in the Rosetta project [29].

Thus, the term software-intensive is used to emphasize that it is this heterogeneous, open, networked and dynamic mix that is in effect, and that the activity of debugging a program is performed in the same way as before. For a large group of developers, this change can be subtle yet dramatic in the sense that actively maintaining the implementation of some complex algorithms is not the direct focus or task[4] anymore. Instead, the activity focuses on piecing together a larger processing system from a wide array of frameworks and third party components. This is similarly dramatic for those that supply developers with these platforms and frameworks.

5.3.1 The Feature Phone, the Smart Phone and the Wardrobe

Current generation smart phones are interesting from a debugging perspective in that they quite cleanly illustrate an embedded to software-intensive 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, but that still perform a clear and distinct function (primarily enabling real-time mobile voice communication between users). On the other hand, there are smart phones in the sense of semi-open[5] generic mobile computing platforms where the distinguishing features from other devices such as tablets or netbooks are related to form factor, user input methods and other superficial details, rather than the computing as such.

[4]These implementations come as neat third party libraries. Thus, the immediate challenges instead concern interfaces and data formatting.

[5]Semi-open because even though there is more openness to third party developers, these devices are still subject to varying degrees of vendor lock-in along with type approval and other certification processes.

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 virtualized through paging and elaborate Memory Management Units (MMUs) and therefore lack normal process separation. A wild pointer could, for instance, corrupt long chains of code and data for 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 malfunction that results in a returned device). Furthermore, the high number (millions) of instances of the same system means that problems with very low repeatability rates will need to be investigated.

The smart phone transition involved large changes of 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 both open and known (parts of Android [72] as well as XNU/Darwin are subject to various open source licensing terms) and developers are openly encouraged to develop and release their software on respective platforms as a means to increase market shares. Subsequently, the end-user able to fine-tune and customize the contents to their hearts' desire. However, these changes and the following economic incentive 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, i.e. obfuscation and DRM [73]. Such measures, however, further complicates debugging.

Now, the crucial aspect is that the developers interact through hierarchies of platforms, some without clear service-level agreements. Thus, it is rather a pre-branded and themed access to these platforms that is being mediated and ultimately sold. The subsequent perceived openness is thus based on how much of the theme that some stakeholder is able to configure. Even though the developers behind individual brands have little influence in configuration of any specific instance, it is ultimately part of their responsibility to optimize the end-user experience.

5.3.2 Remarks

The consequences of the SiS transition in respect to debugging is brutal since this transition advocates a setting in which fundamental mechanisms are being purposely counteracted, giving the analyst limited access and insight into the inner workings of both modern and legacy components. The immediate consequence is that the parts that an analyst is capable of instrumenting are limited by both technical, political and judicial barriers.

5.4 Moving Forward

The situation is thus that debugging efforts for central, and to some extent critical, systems are made more challenging and will put increased pressure on analysts that are already facing tough challenges. Thus, improvements to two key categories, tracers and experiment environments, are suggested as a means for advancing the interplay between development, maintenance and tools. For a discussion on experiment environments, please refer to Chapter. 6.

5.4.1 Systemic Tracing

A lot of development in regards to debugging tools revolves around optimizing the implementation of the respective roles (particularly for symbolic debuggers) towards the goals of minimizing overhead and compensating for a growing disparity between source-code descriptions and the executing code that is actually instrumented. The ideal is that the techniques involved should work equally well for a standard build (merely complemented with a database of symbols), eliminating the need for special debug builds.

A major development in terms of tracing is the expansion of (virtual-) machine support in the direction of tracing frameworks [30]. Recent works that does this include dtrace [31], systemtap [74], PinOS [32], and Lttng [33]. The basic idea is that the code is prepared with *instrumentation points*. These can be implemented through, for instance, No OPeration (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 [75] and the lttng's viewer, the actual presentation consists of fairly simple 2D graphs and histograms, similar to those used by many profilers.

5.4.2 Trace-probe Networks

The suggested enhancement to tracing frameworks, partly concerns loosening the grip and focus on the specifics of how each individual trace probe is managed. 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 strict control is not always possible. 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.

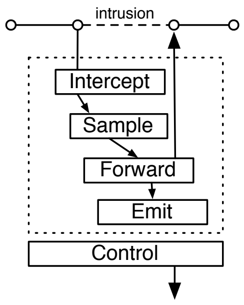


Figure 5.1: Key actions for a trace probe.

When working towards such goals, we begin by outlining a more abstract probe, illustrated in Fig.5.1, and breaking it down into a few key functions.

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 *intrusion*. It should always be a goal to keep this as low as possible. Intrusion can be measured in a sense by the time (or number of state changes) that passes from the point of interception to when control has been returned to the target. Interception as such can be implemented in numerous ways, and it is crucial to select the option that is believed to, again, impose the least intrusion. Preexisting debugging interfaces can, of course, still be used to obtain 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 [76], as well as the interface for process or raw memory access, specialized drivers, loadable plugins, JIT facilities of virtual machines [34], exploiting security holes that allow for remote code injection, 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.) and packaging (compression, encryption).

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.

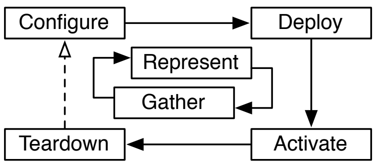


Figure 5.2: Coordinating a network of trace probes.

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. 5.2. 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 ring (*configure*, *deploy*, *activate*, *teardown*) and an inner 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 point, it is also 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.

The fact of the matter is that this idea has already been realized in a number of other context. 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. HP's OpenView is one example of a long standing such tool, but there are interesting free-tools as well, such as smokeping [77]. Surveillance systems for critical infrastructures such as the power grid have also had a similar designs for a long time, even if these designs have been static.

Other real-world examples of distributed probe networks are the command and control structures used by botnets[6]. A control structure that is powerful enough to invade the privacy of millions of people and capable of directing large-scale

[6]Large networks of hijacked personal computers used for gathering information (espionage), performing distributed attacks, send spam, etc.

attacks on the information infrastructure of large corporations surely has aspects that could be leveraged for more productive uses, like debugging.

The next step, when the immediate technical challenges are dealt with, is to alter the probes so that they can also perform interventions (trace-actor networks), thus bridging a key feature of the symbolic debugger. By combining trace-actor networks and refined experiments, we might finally get a setting where it is possible to experimentally develop and improve frameworks for causal reasoning in software error analysis.

5.5 Conclusions

To briefly summarize the tools and their respective drawbacks:

With the *symbolic debugger*, the level of control and intervention required to support breakpoints and 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) or more dubious (worms, viruses) programs take advantage of this fact to alter their behavior in, for the analyst, counter-productive ways.

With *tracers* the central issue is the model to which gathered measurements are attached. Even though this is a challenge that 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 that can be extracted and made useful from an instance that has come to a very distinct halt. In the case of tracing, you instead have a series of tools that provides small samples of specific key data, which has often been generated throughout the life-span of the program in question. The most primitive of these are the ones that are integrated in the subject, e.g. *printf statements* or calls to *system log facilities*.

For *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 state holders necessary to perform the analysis gets overwritten at a rapid rate. Thus, post-mortem analysis depends on how much of the type and location of specific data on how can be determined in advance, but also that the underlying causes trigger a snapshot for analysis proximate to when the effect in question occurred.

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).

The systemic shift illustrated in Sec. 5.3 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.

Thus, the grander challenge ahead is not only to fashion a well-coordinated set of modern tools and a representative experiment environment where these tools can be reliably used, but also to *teach and train* debugging (or try and integrate such training in other teaching endeavors) in order produce a greater number of analysts skilled enough to tackle the challenge.

5.6 Errata

The following alterations have been made during the preparation of this thesis:

- The section on *Experiment Environments* was replaced with a reference to Chapter 6.

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 experimentations. 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) [78] and the (SEESGEN-ICT) thematic network [79]. 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 [80] efforts (Fig. 6.1). A contrasting frame of reference between the work in this thesis and in GENI may thus be found through planetlab [35], but this mainly concerns scale and application domain specificity.

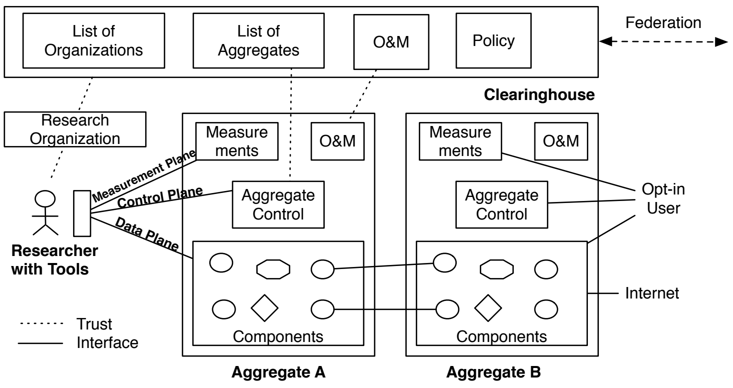


Figure 6.1: NSF GENI Architecture.

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 paper will expand upon. It covers three distinct parts: The physical distribution network, the agent logic and sensoring 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 a of

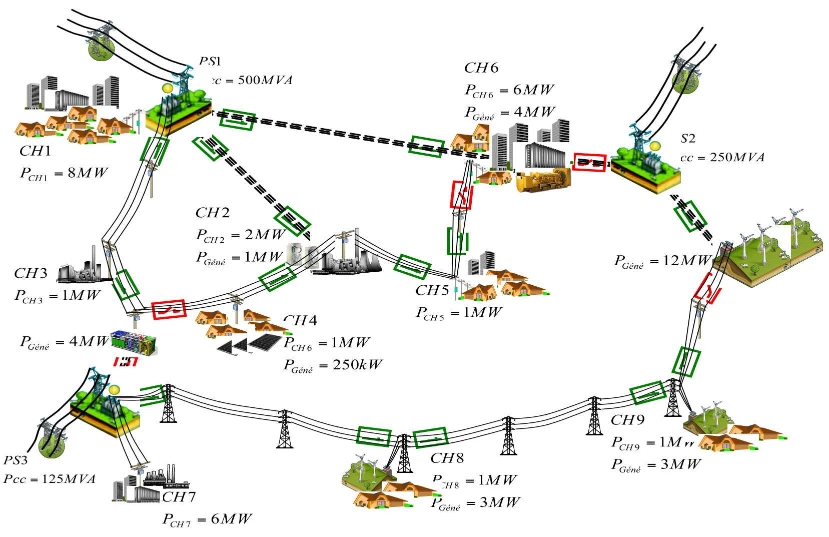


Figure 6.2: A model of the microgrid.

30 MVolt Ampere (VA) rated power at 20 kV. In order to best represent the behaviour 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 behaviours of the network, including aggregated loads, network components (on-load tap changers for instance) and Distributed Energy Resources (DER)/Renewable Energy Sources (RES). A strong requirement is that the performance of the control systems must be unchanged in comparison with the real system. 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 [81] and MATLAB (OPC) toolbox [82]. 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 [54]/ (Modbus) [55]) 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 [83]) 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 Distributed System/Service Operator (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 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.

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 [84] 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, such as the Programmable Logic Controllers (PLCs), is handled by a component called the *Supervisor Communication Manager*. The communication technologies involved include OPC, the native driver equipment protocol, DDE [56], 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 PLCs.

[1]Flair 200C, fault passage indicators furnished by Schneider Electric Telemecanique.

[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.

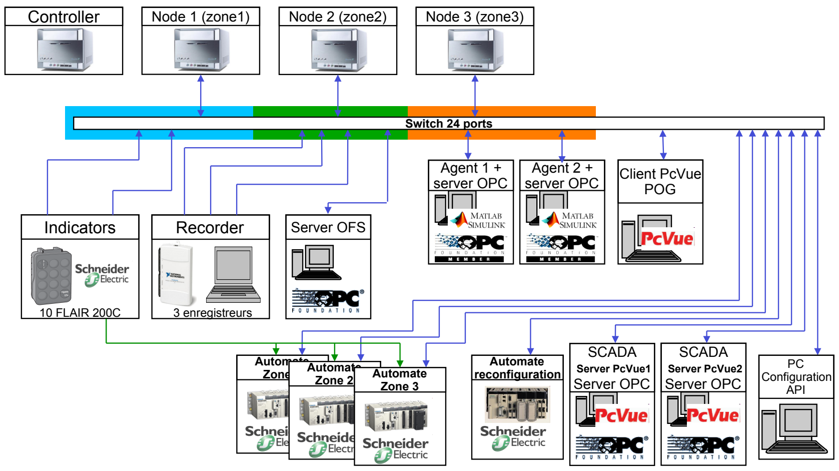


Figure 6.3: Early overview of the ICT solution. Note that parts of the base ICT layer are not covered by corresponding monitoring nodes.

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 ordered 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 behaviour 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 [8]. For the other category, *dynamic abstractions and separations*, observable computation patterns, i.e. behaviours, 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[5] which still influence the execution in such a way that

[4]Even so, the limitations that stem from the age-old computability 'halting problem' still apply.

[5]In 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.

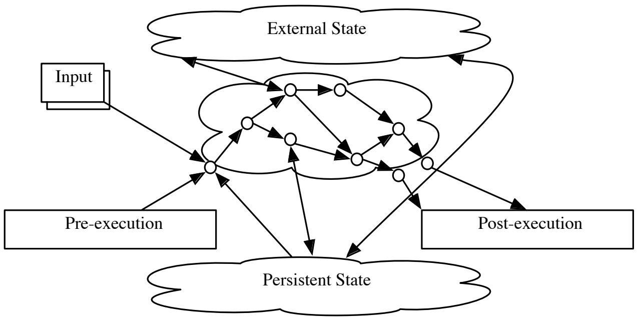


Figure 6.4: The virtualization problem.

the protective enclosure can be breached (means to intentionally do this is a currently an active area of research within software security)[6]. 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.

As an approach to expanding this control to incrementally larger borders, a solution called `EXP` [19], was developed as part of previous projects [85]. 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.

[6]This is a rather brief and shallow summary of the problem. The full extent of this discussion is, however, well outside the scope of this chapter.

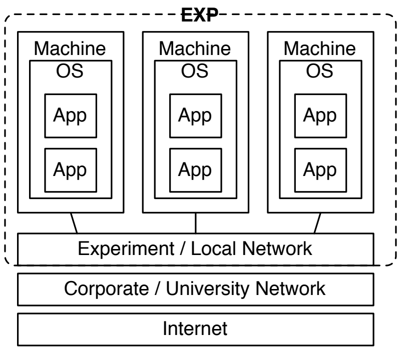


Figure 6.5: EXP-I and its respective borders.

- *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.

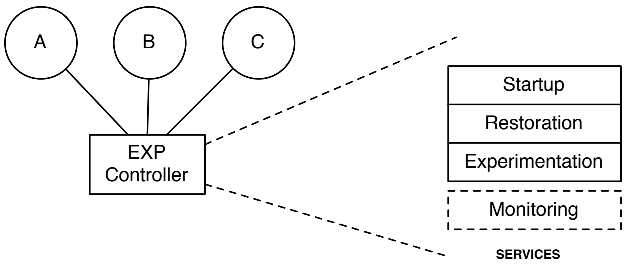


Figure 6.6: EXP-I services, monitoring added in EXP-II.

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[7]

[7]In some cases this includes their power supply using programmable outlets.

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.

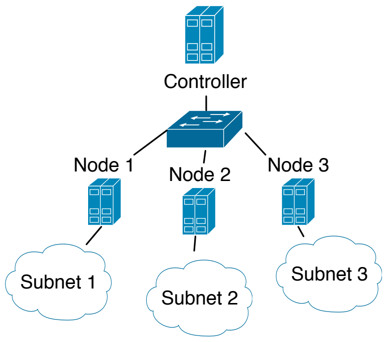


Figure 6.7: The basic lab setup.

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) [57] 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.

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.

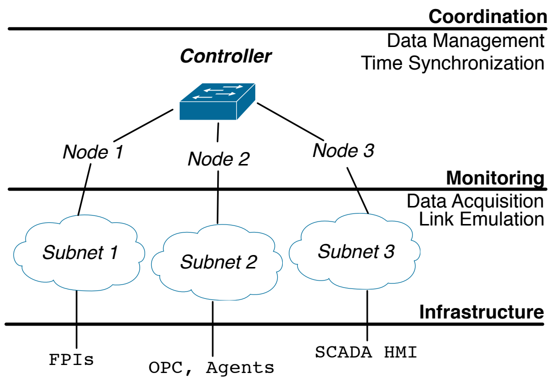


Figure 6.8: The basic lab instantiated, one mode of operation.

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 [86] operating system, configured as simple dummynet [36] 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.

6.4.2 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`$\rightarrow$ (site a, node 1) $\rightarrow$(site a, controller) $\rightarrow$ (site b, controller) $\rightarrow$ (site b, node 2) $\rightarrow$ (site b, controller) $\rightarrow$ (site a, controller) $\rightarrow$ (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.

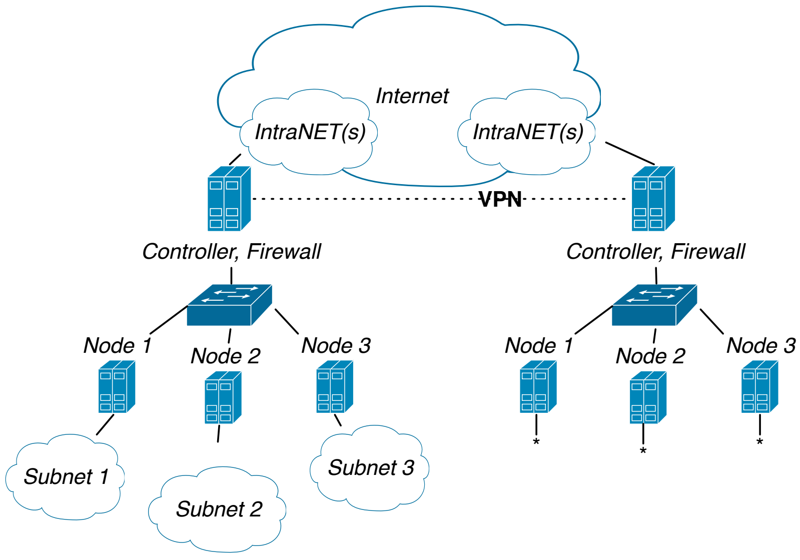


Figure 6.9: Unified Operation Overview.

6.5.1 Technical Barriers

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 [87, 88], and since the nature of software behaviour 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

[8]More specifically, far from all processing circuitry have clearly defined and tested reactions to arguments received as part of communication protocols.

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.

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[9] 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 [21] 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) [22].

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

[9]The 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.

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) [58] 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.

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 [59, 60], 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. Provided, for instance, the development of more resilient grid structures like microgrids, the SCADA and the end-users will become even more interesting and likely targets simply because of the ability to more precisely guide an attack. 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.

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.

6.8 Errata

The following alterations have been made during the preparation of the thesis:

- Figure depicting basic SCADA and the corresponding description were moved to the *Context* chapter, Page 19.
- A superfluous figure on how the OPC subsystem related to Fig. 6.7 was removed.

7 Conclusions

In this, the final chapter of the thesis, we summarize the efforts and contributions thus far and contrast them with relevant criticism, related work and validation and we also explore possible roads for future work.

7.1 Summary

Starting with the mission statement *"to enable the transition of brittle software-intensive infrastructures into resilient software-intensive infrastructures"*, we began our work by addressing the subsequent research questions:

- 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, *(Paper I)*. 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*. 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, so that the machine space of a particular virtualization may be the virtual space of some other virtualization.

We also established the constraints of virtualization, based on whichever benefits that are currently desired for the targeted system. From this set-up, a series of principles was deduced, i.e. in order to maintain and ascertain the validity of each placed virtualization 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. The success of this relies on establishing the external state holders that influence and hinder controlled experimentation with a subject (*Fig. 7.1*). The principles can be applied iteratively, essentially allowing for adaptive dynamic hardening.

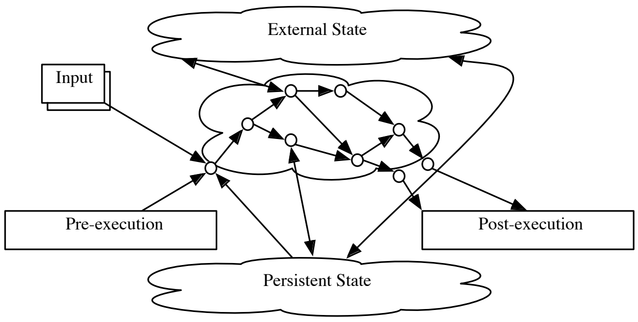


Figure 7.1: External state-holders and their influence on virtualized resource(s).

We also provide a basis for how to experiment with these principles on a live subject. Working from the other stated research question:

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

We examined the available tools currently widely used in both industry and the academy, and the respective mechanisms on which said tools operate, and reasoned as to how these mechanisms may interplay with the aforementioned principles. Some of the major shortcomings that were noted include:

1. With the *symbolic debugger*, the level of control and intervention required for implementing *breakpoints*[1], 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.
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*.

Many software systems that previously had well-defined or isolated roles (such as cellphones, games, browsers and even web services), turn into more open frameworks that allow third party developers in as time goes by and as the popularity of the system in question grows. This makes matters much more complicated for those responsible for the platform. Many facilities for allowing

[1]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.

users to report problems, suggest improvements, etc. do not, and in some cases cannot, include the third party developers in question. As the control and influence over how the services rendered will be perceived lessen, the challenge in using the aforementioned tools increases. This is in part due to hidden interactions by third-party components (amplified by the use of obfuscation, digital rights management and other means for limiting or hindering reverse engineering and similar activites), but also since the stakeholders responsible for analysis have fewer options for quickly understanding the function and implementation details of some components (lack of access to source-code, build system configuration, design documentation etc.).

To start addressing these concerns, it was suggested that a more broad control of the tool chain and the interaction between individual tools is needed, but also that current tool suites are, to an extent, insufficient and need to be complemented by a broader framework of configurable probes capable of alternating between both native and non-native forms of representing measurements.

Finally, as a step towards experimentally validating these ideas in the context of critical software-intensive infrastructures, a distributed ICT experimental environment (*Fig. 7.2*), `EXPII` (*Paper III*) was engineered following the final 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, and which guidelines should regulate such experimentation?*

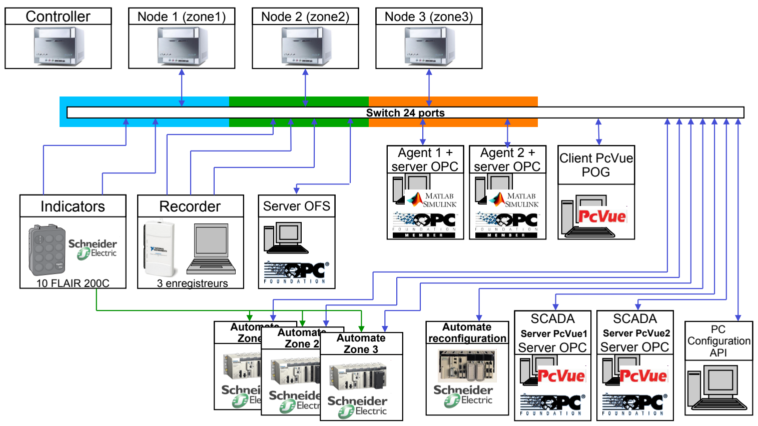


Figure 7.2: Overview of ICT components and their interconnections.

The developed environment currently enables us to virtualize, monitor and intervene with, amongst 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).

7.2 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 this thesis. The major works that were taken into account, rather than simply the references used, will be covered briefly.

In terms of European research efforts, the projects directly related to this thesis are (CRISP) [85], INTEGRAL [85] and SEESGEN-ICT [79].

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 RES penetration. The project was concluded in 2006 and, among the deliverables, D3.1 [89] and D5.3 [90] are related to the design and implementation of an experiment environment that was in fact the predecessor to the one described in Paper III.

INTEGRAL

Integrated ICT-platform based Distributed Control in Electricity Grids has the cited objective of building and demonstrating an industry- quality reference solution for 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. Furthermore, the demonstrator is likely to forego large restructuring in the near future.

The interplay beteween the factors can be found in Sect. 7.3.

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 associated requirements and barriers. SEESGEN-ICT aims to produce policy recommendations, identify best practices and draw scenarios and roadmaps 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 preliminary results, parts of the deliverables D3.1, D3.2, D3.3 and D3.4 cover many of the discoveries made here, but within a more generic smart-grid context.

7.3 Validation

In this section, we examine how the respective research questions connect to the contributed principles, tools and environments, and how they all combine into a demonstrator used to show a self-healing response to a disturbance within a cell of a scaled down microgrid. Data from initial experiments are also presented.

Expanding on the principles from *(Paper I)*, it is clear that they are indeed very similar to the ones that enable resilience in a broader sense as presented in the introductory chapter. To make a brief comparison:

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.

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 [38] 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 and to network communication, one of the main points of the demonstrator was to illustrate viability. Self-healing software as such is excluded from the scope of this thesis and therefore, this principle does not currently have any corresponding virtualization principles. However, even with the demonstrated self-healing of the grid, this needs to be considered from the perspective of a larger, aggregate system and not from the individual disciplines as such.

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. 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.

[2]In execution, this concerns storing and replaying all interaction that occurs within the timeframe from the last accessible snapshot and the detected error.

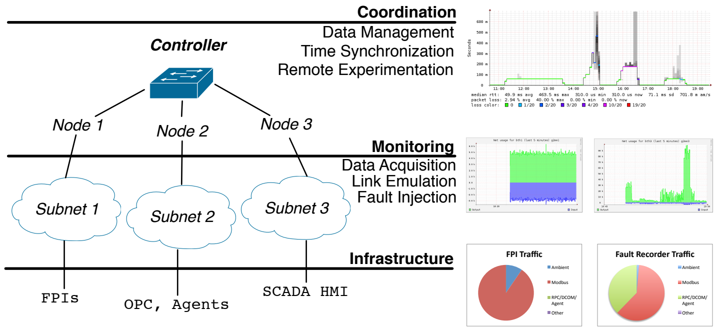


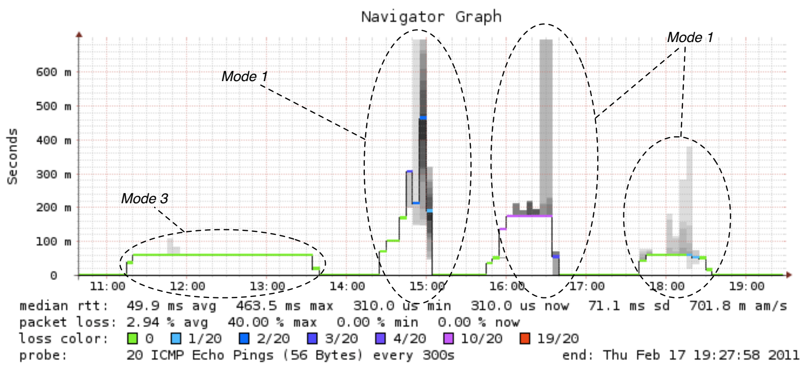
Figure 7.3: Snapshots of the monitoring present in the demonstrator (SCADA HMI screenshot excluded).

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 error detection but also in order to find reconfigurations that could invalidate previous efforts.

The monitoring used (Fig. 7.3) 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 generating 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 [77]).

The traffic logs enabled post-mortem analysis when combined with tools for that purpose (the results presented are from the use of Wireshark [91]). 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.



```
median rtt:   49.9 ms avg   463.5 ms max   310.0 us min   310.0 us now   71.1 ms sd   701.8 m am/s
packet loss:  2.94 % avg   40.00 % max   0.00 % min   0.00 % now
loss color:   🟩 0   🟦 1/20   🟦 2/20   🟦 3/20   🟪 4/20   🟪 10/20   🟥 19/20
probe:        20 ICMP Echo Pings (56 Bytes) every 300s                     end: Thu Feb 17 19:27:58 2011
```

Figure 7.4: Latency traces for several iterations of a self-healing sequence.

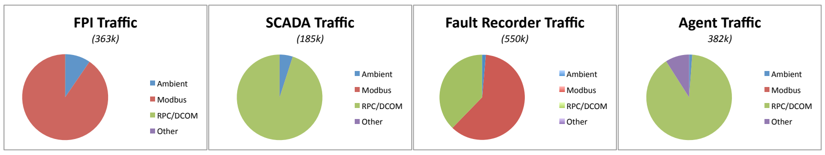


Figure 7.5: Protocols (Ambient, Modbus, RPC/DCOM, Other) in proportion to the total traffic of each subnet.

7.3.1 Results

The final environment is similar to the one illustrated in Fig. 7.2, 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. 7.4 depicts the latencies of the SCADA subnet over the course of a day of experimentation. *Mode 3* corresponds to remote experimentation, 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 grey indicate the variance between probes. The latency probes were configured to pass 20 probes every 5 minutes.

Fig. 7.5 depicts the distribution of the detected protocols in each subnet relative

Source	Destination	% of total traffic
FPI	FR	22
FR	FPI	25
FR	Agent	26
Agent	FR	5
Agent	SCADA	10
SCADA	Agent	12

Table 7.1: Traffic ratios.

FPI	FR	Agent	Scada	Time Elapsed
(0) (3/6)	(0) (10/167)	(0) (130/21)	(0) (25/25)	6s
(25) (5/3)	(25) (4/28)	(25) (30/4)	(25) (4/5)	24s
x (3/2)	x (2/0)	x (20/1)	x (3/3)	36s

Table 7.2: Notable link configurations (latency introduced, milliseconds), peak traffic rates (output/input) KiB/s and time for the self-healing scenario to complete.

to the total amount of traffic within that subnet, extracted from the snapshots of the raw traffic logs (libpcap [92] 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. 7.3.1 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.

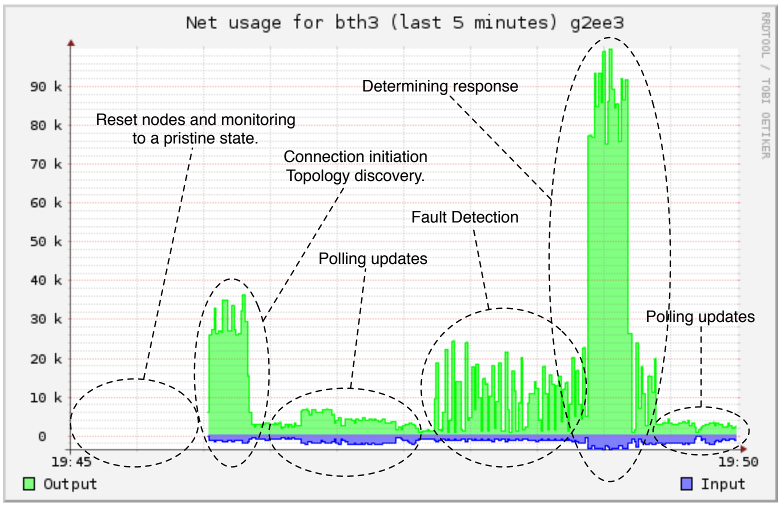


Figure 7.6: Annotated graph of one iteration of a self-healing scenario, from the perspective of the node governing the agent. Thus, output entails data being sent into the agent subnet.

Table. 7.3.1 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 nationality, regulations, fault-type and other factors) is around 160s. Considering the early stages of agent development, the SCADA was set up to ignore the issued command because of the serious risk of damaging equipment if the wrong action was initiated.

Lastly, Fig. 7.6 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. 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.
- The communication protocols involved are poor choices in respect to establishing protective devices within the confines of the SCADA. Furthermore, the higher level protocols couple with lower level addressing schemes, preventing Network Address Translation (NAT) and other optimizations (a transition to IPv6 would, for instance, not be possible without extensive modifications).
- Except for an operator actively browsing around with the HMI complaining about the interface feeling less responsive, there were no notifications in the HMI of problems pertaining to changes in the network configuration. The transition between running locally and redirecting all traffic through several foreign networks was, in that sense, transparent.
- Preliminary tests indicate that application- layer checksum facilities were in part not used, and in part ignored.

7.3.2 Future Work

The work presented is to a large extent open ended with several interesting paths left to explore.

The need for well-engineered and open tools for enabling controlled experimentation on software-intensive systems is still strong. Thus, the key may well be to open up the environment and the underlying tools to a select few application domains in addition to that of critical infrastructures, and then focus on retrofitting and generalizing developments from these domains into a shared package. Some application domains could include forensics and e-learning labs since they face similar challenges in terms of monitoring and supervised control.

For the current environment, there are already many direct enhancements and expansions to consider. Based on the initial tests, it is quite clear which protocols, or rather, which specific data that need to be communicated, and how sensitive and important this data are in regards to the ongoing self-healing process. It would, however, be interesting to further verify this against competing SCADA solutions and then evaluate the result in the light of impending smart grid deployment in general and the consequences of expanding the SCADA monitoring approach towards more low-voltage applications (such as smart homes) in particular.

From a security perspective, there is a large ongoing discussion on how to retrofit various kinds of protective measures that are otherwise common parts of corporate IT. This discussion is further fueled by recent examples of just how vulnerable SCADA systems can be, and the considerable impact of successful attacks, suggesting that merely relying on encrypting and tunneling sensitive data may well be unwise. But the investment of the full scale option – in terms of deploying and maintaining public-key encryption, tuning intrusion detection systems, undertaking procedures for deploying patches and revoking access keys and training personnel, etc – is considerable and it should therefore be worth looking into developing attack scenarios against which these protective measures can be exercised and evaluated.

Turning towards slightly more academic topics, it would be interesting to combine the tracing approaches from software debugging with a more SCADA- like perspective to gain a larger toolbox of sensors and sensor network configurations capable of creating dynamic monitoring tools for studying and correlating events on many levels (combining adaptive traces from surveillance equipment, network traffic data, infrastructure sensor information, etc.), allowing us to better monitor infrastructure ”health” and investigate anomalies. This notion has further branches in the inherent conflict of interest between, on the one hand, information security and, on the other, maintenance and debugging.

A concern shared by all these topics, is the grand challenge of educating new generations of analysts, and to retrain experienced analysts towards deeper understanding of modern software dynamics in relation to a wider software-intensive perspective. This might be attainable through minor incisions to current study programs, but chances are that a larger restructuring may be necessary. Indications as to the how and the why behind such changes can be found in [93].

A Glossary

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

Definitions

Software/Program A (computer) program generally refers to a specific series of instructions to a computer. Each instruction has two possible parts, code (mandatory) and data (optional, depends on the specific instruction). Software is used as a broader term for collections of programs with a designated purpose or task.

State The current operating conditions of a computer or a program.

State-holder Any data that current, and future, computing depend on.

Execution The activity of carrying out the instructions of a program.

Environment All the parts or programs involved in the upkeep, management and execution of some specific piece of software.

Static Programs that cannot, or do not, change during execution.

Dynamic Programs that change during execution. Can also refer to information that is created during execution.

Behaviour Specific or generic patterns on observable and measurable reactions as a consequence to some stimulation or activity.

Machine Any device (physical or abstract) that enables computing (any or all of the three categories: storage, communication and calculation).

Virtualization A program that dynamically translates data into code native to a machine, allowing programs to be articulated and executed in two different levels, in a virtual space and in a machine (or native) space.

Interface The dimensions and boundaries of data exchange.

Protocol The regulated flow of information across an interface.

Embedded System A combination of software and computer(s) that is designed to fit a very specific and well-defined role or function.

Resilience The ability of a system to harness disturbances.

Fault-tolerant The ability of a system to continue operation in the event of the failure of some of its components.

Dependable The extent of which a system can be relied on or trusted.

Brittle The tendency of a system to break when subject to high stress.

Acronyms

API	Application Programming Interface
CISC	Complex Instruction Set Computer
COTS	Commercial, Off-the-Self
CPU	Central Processing Unit
GPU	Graphics Processing Unit
MMU	Memory Management Unit
NOP	No OPeration
DMA	Direct Memory Access
DSP	Digital Signal Processor
FPI	Fault Passage Indicator
FR	Fault-Recorder
SCADA	System Control and Data Acquisition
IDS	Intrusion Detection System
ICT	Information and Communication Technologies
IPS	Information Processing System
IS	Information System
MTU	Main Terminal Unit
RTU	Remote Terminal Unit
HMI	Human Machine Interface
ROP	Return- Oriented Programming
SQL	Structured Query Language
LV	Low Voltage
MV	Medium Voltage
HV	High Voltage
UNIX	Uniplexed Information and Computing System
OSI	Open Systems Interconnection
PRNG	Pseudo- Random Number Generator
RAM	Random Access Memory
RFID	Radio Frequency Identifier
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
TCP	Transmission Control Protocol
IP	Internet Protocol
V	Voltage
DER	Distributed Energy Resources
RES	Renewable Energy Sources
RAID	Redundant Array of Inexpensive Disks

IT	Information Technology
NSF	National Science Foundation
VA	Volt Ampere
INTEGRAL	Integrated ICT-platform based Distributed Control in Electricity Grids
GENI	Global Environment for Network Innovations
OFS	OPC Factory Server
VPN	Virtual Private Network
KVM	Keyboard, Video and Mouse
WORE	Write Once, Rune Eveywhere
DSO	Distributed System/Service Operator
LSO	Local System/Service Operator
DG	Distributed Generation
JNI	Java Native Interface
VM	Virtual Machine
IED	Intelligent Electronic Device
WAN	Wide Area Network
LAN	Local Area Network
VLAN	Virtual Local Area Network
UDP	User Datagram Protocol
PLC	Programmable Logic Controller
ARP	Address Resolution Protocol
NTP	Network Time Protocol
SNMP	Simple Network Management Protocol
STP	Spanning Tree Protocol
MAC	Media Access Control
NAT	Network Address Translation
DCOM	Distributed Common Object Model
RPC	Remote Procedure Call
JVM	Java Virtual Machine
DRM	Digital Rights Management
HLE	High-Level Emulation
OS	Operating System
GNU	Gnu is Not Unix
MAME	Multiple Arcade Machine Emulator
WINE	Wine Is Not an Emulator
LLVM	Low-level Virtual Machine

B References

Articles

[1] D. Patterson and G. Gibson, "A case for redundant arrays of inexpensive disks (raid)," *Proceedings of the 1988 ACM*, Jan 1988.

[2] K. Thompson, "Reflections on trusting trust," *Communications of the ACM*, Jan 1984.

[3] E. Coffman and M. Elphick, "System deadlocks," *ACM Computing Surveys*, Jan 1971.

[4] B. Stahl, L. L. Thanh, R. Caire, and R. Gustavsson, "Experimenting with infrastructures," in *Critical Infrastructure (CRIS), 2010 5th International Conference on*, September 2010, pp. 1 –7.

[5] R. Gustavsson and B. Ståhl, "Self-healing and resilient critical infrastructures," in *CRITIS*, ser. Lecture Notes in Computer Science, R. Setola and S. Geretshuber, Eds., vol. 5508. Springer, 2008, pp. 84–94.

[6] P. Mellstrand and B. Ståhl, "Analyzing systemic information infrastructure malfunction," in *Critical Infrastructures, 2009. CRIS 2009. Fourth International Conference on*, april 2009, pp. 1 –4.

[7] R. Gustavsson and B. Ståhl, "The empowered user - the critical interface to critical infrastructures," in *Critical Infrastructure (CRIS), 2010 5th International Conference on*, September 2010, pp. 1 –3.

[8] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, pp. 1–84, 2010.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 164–177, October 2003. Available: http://doi.acm.org/10.1145/1165389.945462

[10] S. Crosby and D. Brown, "The virtualization reality," *Queue*, vol. 4, pp. 34–41, December 2006. Available: http://doi.acm.org/10.1145/1189276.1189289

[11] D. A. Menascé, "Virtualization: Concepts, applications, and performance modeling," in *Int. CMG Conference*. Computer Measurement Group, 2005, pp. 407–414.

[12] M. Arnold, S. Fink, D. Grove, and M. Hind, "A survey of adaptive optimization in virtual machines," *Proceedings of the IEEE*, pp. 449–466, Feb 2005.

[13] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, ser. CGO '04, 2004, pp. 75–.

[14] 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.

[15] G. Phillips, "Simplicity betrayed," *j-CACM*, vol. 53, no. 6, pp. 52–58, jun 2010.

[16] G. James, B. Silverman, and B. Silverman, "Visualizing a classic cpu in action: the 6502," in *ACM SIGGRAPH 2010 Talks*, ser. SIGGRAPH '10, 2010, pp. 26:1–26:1.

[17] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS '10, 2010, pp. 559–572.

[18] M. Bishop and D. Frincke, "Who owns your computer? [digital rights management]," *Security Privacy, IEEE*, vol. 4, no. 2, pp. 61 –63, March 2006.

[19] P. Mellstrand and R. Gustavsson, "Experiment based validation of ciip," *Lecture Notes in Computer Science*, vol. 4347, pp. 15–29, 2006.

[20] A. Barth, C. Jackson, and J. C. Mitchell, "Securing frame communication in browsers," *Commun. ACM*, vol. 52, pp. 83–91, June 2009.

[21] 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.

[22] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "An Analysis of Data Corruption in the Storage Stack," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.

[23] J. Shore, "Fail fast," *IEEE Software*, vol. 21, pp. 21–25, 2004.

[24] K. Araki, Z. Furukawa, and J. Cheng, "A general framework for debugging," *IEEE Softw.*, vol. 8, pp. 14–20, May 1991.

[25] M. N. Gagnon, S. Taylor, and A. K. Ghosh, "Software Protection through Anti-Debugging," *Security & Privacy Magazine, IEEE*, vol. 5, no. 3, pp. 82–84, 2007.

[26] M. Mateas and N. Montfort, “A box, darkly: Obfuscated code, weird languages, and code aesthetics,” *Proceedings of the 2005 Digital Arts and Culture Conference*, pp. 144–153, 2005.

[27] H. Xu and S. J. Chapin, “Address-space layout randomization using code islands,” *J. Comput. Secur.*, vol. 17, pp. 331–362, August 2009.

[28] P. T. Zellweger, “An interactive high-level debugger for control-flow optimized programs,” *SIGPLAN Not.*, vol. 18, pp. 159–172, March 1983.

[29] S. Bansal and A. Aiken, “Binary translation using peephole superoptimizers,” in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI’08, 2008, pp. 177–192.

[30] D. Toupin, “Using tracing to diagnose or monitor systems,” *IEEE Softw.*, vol. 28, pp. 87–91, January 2011.

[31] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, “Dynamic instrumentation of production systems,” in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC ’04. Berkeley, CA, USA: USENIX Association, 2004, pp. 2–2.

[32] P. Bungale and C. Luk, “Pinos: a programmable framework for whole-system dynamic instrumentation,” *Proceedings of the 3rd international conference on Virtual execution environments*, pp. 137–147, 2007.

[33] P.-M. Fournier, M. Desnoyers, and M. R. Dagenais, “Combined tracing of the kernel and applications with LTTng,” in *Proceedings of the 2009 Linux Symposium*, jul 2009.

[34] M. Olszewski, K. Mierle, A. Czajkowski, and A. D. Brown, “Jit instrumentation: a novel approach to dynamically instrument operating systems,” *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 3–16, March 2007.

[35] L. L. Peterson, A. C. Bavier, M. E. Fiuczynski, and S. Muir, “Experiences building planetlab,” in *OSDI*. USENIX Association, 2006, pp. 351–366.

[36] M. Carbone and L. Rizzo, “Dummynet revisited,” *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 12–20, April 2010.

Books

[37] E. Hollnagel, D. Woods, and N. Leveson, *Resilience Engineering: Concepts And Precepts*. Ashgate Publishing, Apr 2006.

[38] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, October 2005.

[39] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, 1st ed. Addison-Wesley Professional, 2009.

[40] R. Fernando, *Graphics Pipeline Performance*. Addison-Wesley Professional, 2004.

[41] J. R. Levine, *Linkers and Loaders*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

[42] Y. N. Srikant and P. Shankar, Eds., *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2002.

[43] C. Cifuentes, *Reverse Compilation Techniques*, 1994, p. 56.

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

[45] R. C. Metzger, *Debugging by Thinking: A Multidisciplinary Approach*. Digital Press, 2003.

[46] L. Simone, *If I Only Changed the Software, Why is the Phone on Fire?: Embedded Debugging Methods Revealed: Technical Mysteries for Engineers*. Newnes, 2007.

[47] J. Pearl, *Causality: Models, Reasoning and Inference*, 2nd ed. New York, NY, USA: Cambridge University Press, 2009.

[48] J. Woodward, *Making Things Happen: A Theory of Causal Explanation*. Oxford University Press, 2003.

[49] J. B. Rosenberg, *How debuggers work: algorithms, data structures, and architecture*. New York, NY, USA: John Wiley & Sons, Inc., 1996.

Standards

[50] "Ieee standard classification for software anomalies," *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pp. C1 – 15, 2010.

[51] ISO, *ISO/IEC 7498-1: Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model*, International Standards Organization ISO, 1994.

[52] "Java native interface specification." Available: http://download.oracle.com/javase/1.5.0/docs/guide/jni/spec/jniTOC.html

[53] J. Postel, "DoD standard Internet Protocol," RFC 791, Internet Engineering Task Force, Sept. 1981. Available: http://www.ietf.org/rfc/rfc791.txt

[54] ——, "DoD standard Transmission Control Protocol," RFC 761, Internet Engineering Task Force, Jan. 1980. Available: http://www.ietf.org/rfc/rfc761.txt

[55] "Modbus protocol reference guide." Available: http://www.modbus.org/docs/PI_MBUS_300.pdf

[56] "About dynamic data exchange." Available: http://msdn.microsoft.com/en-us/library/ms648774.aspx

[57] "Local and metropolitan area network standards." Available: http://standards.ieee.org/getieee802/download/802.1Q-2005.pdf

[58] J. Postel, "User Datagram Protocol," RFC 768, Internet Engineering Task Force, Aug. 1980. Available: http://www.ietf.org/rfc/rfc768.txt

Online Resources

[59] M. Brunner, H. Hofinger, C. Krauß, C. Roblee, P. Schoo, and S. Todt, “Infiltrating critical infrastructures with next-generation attacks: W32.stuxnet as a showcase threat,” Fraunhofer SIT, Darmstadt, dec 2010. Available: http://publica.fraunhofer.de/documents/N-151330.html

[60] “Global energy cyberattacks: Night dragon,” Whitepaper, feb 2011. Available: www.mcafee.com/us/resources/white-papers/wp-global-energy-cyberattacks-night-dragon.pdf

[61] “Final report on the August 14, 2003 blackout in the United States and Canada: Causes and recommendations,” April 2004. Available: https://reports.energy.gov/

[62] E. Bachaalany, “An attempt to reconstruct the call stack.” Available: http://www.hexblog.com/?p=104

[63] “Multiple arcade machine emulator.” Available: http://www.mamedev.org

[64] “Wine is not an emulator.” Available: http://www.winehq.org

[65] “Gnu ld linker.” Available: http://www.gnu.org/software/binutils

[66] T. D. Raadt, “Exploit mitigation techniques,” 2005. Available: http://www.openbsd.org/papers/ven05-deraadt/index.html

[67] M. Dowd, “Application-specific attacks: Leveraging the action-script virtual machine,” 2007. Available: http://documents.iss.net/whitepapers/IBM_X-Force_WP_final.pdf

[68] A. Sotirov and M. Dowd, “Bypassing browser memory protections,” 2008. Available: https://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf

[69] R. Wojtczuk, “Subverting the xen hypervisor,” 2008. Available: http://invisiblethingslab.com/bh08/papers/part1-subverting_xen.pdf

[70] “Gdb: The gnu project debugger.” Available: http://www.gnu.org/software/gdb

[71] “The dwarf debugging standard.” Available: http://www.dwarfstd.org

[72] “Google android.” Available: http://www.android.com

[73] “Android application licensing, implementing an obfuscator.” Available: http://developer.android.com/guide/publishing/licensing.html

[74] “Architecture of systemtap: a linux trace/probe tool.” Available: http://sourceware.org/systemtap/archpaper.pdf

[75] “Developer tools: Apple developer overview.” Available: http://developer.apple.com/technologies/tools/

[76] “Embedded elf debugging.” Available: http://www.phrack.com/issues.html?issue=63&id=9

[77] T. Oetiker and N. Tyni, “Smokeping.” Available: http://oss.oetiker.ch/smokeping

[78] Integral, ict-platform based distributed control in electricity grids, fp6-038576. Available: http://www.integral-dc.eu

[79] “Seesgen-ict, supporting energy efficiency in smart generation grids through ict, cip-ict psp-2-2008-2.” Available: http://seesgen-ict.rse-web.it

[80] “Geni, the global environment for network innovations.” Available: http://www.geni.net

[81] MATLAB, “version 7.10.0 (r2010a),” 2010. Available: http://www.mathworks.com/products/matlab

[82] “Opc toolbox.” Available: http://www.mathworks.com/products/opc

[83] N. Instruments, “Ni labview.” Available: http://www.ni.com/labview

[84] P. Solutions, “Pcvue.” Available: http://www.arcinfo.com/

[85] “distributed intelligence in critical infrastructures for sustainable power.” Available: http://crisp.ecn.nl

[86] “The freebsd project.” Available: http://www.freebsd.org

[87] “Aurora vulnerability,” whitepaper, 2006. Available: http://unix.nocdesigns.com/aurora_white_paper.htm

[88] D. Maynor and R. Graham, “Scada security and terrorism: We’re not crying wolf,” 2006. Available: http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Maynor-Graham-up.pdf

[89] “Crisp, d3.1 - specification of experiments and test set up.” Available: http://crisp.ecn.nl/deliverables/D3.1.pdf

[90] “Crisp, d5.3 - final summary report.” Available: http://crisp.ecn.nl/deliverables/D5.3.pdf

[91] “Wireshark.” Available: http://www.wireshark.org

[92] “Tcpdump / libpcap.” Available: http://www.libpcap.org

[93] “Introductory computer science education at carnegie mellon university: A deans’ perspective.” Available: http://www.cs.cmu.edu/~bryant/pubdir/cmu-cs-10-140.pdf

ABSTRACT

Software has, for better or worse, become a core component in the structured management and manipulation of vast quantities of information, and is therefore central to many crucial services and infrastructures. However, hidden among the various benefits that the inclusion of software may bring is the potential of unwanted and unforeseen interactions, ranging from mere annoyances all the way up to full-blown catastrophes.

Overcoming adversities of this nature is a challenge shared with other engineering ventures, and there are many developed strategies that work towards eliminating various kinds of disturbances, assuming that it is possible to apply such strategies correctly. One approach in this regard, is to accept some anomalous behaviors as mere facts of life and make sure that the situations experienced are dealt with in an expeditious manner, while at the same time trying to discover, implement and improve safe-guards that can lessen adverse consequences in the event of future problems; in short, to embed resilience.

The work described in this thesis explores the foundations of software resilience, and thus covers the main resilience-enabling mechanisms, along with supporting tools, techniques and methods used to embed resilience. These instruments are dissected and analyzed from the perspective of stakeholders that have to operate on pre-existing, critical, large and heterogeneous subjects that are to some extent already up and running at the point of instrumentation. Finally, in the course of describing this subject, the thesis describes a demonstrator environment for self-healing activities in a partially damaged power grid, its construction details and the initial results of the study conducted in this environment.

BLEKINGE TEKNISKA HÖGSKOLA · BTH ·

ISSN 1650-2140
ISBN 978-91-7295-206-5

2011:05