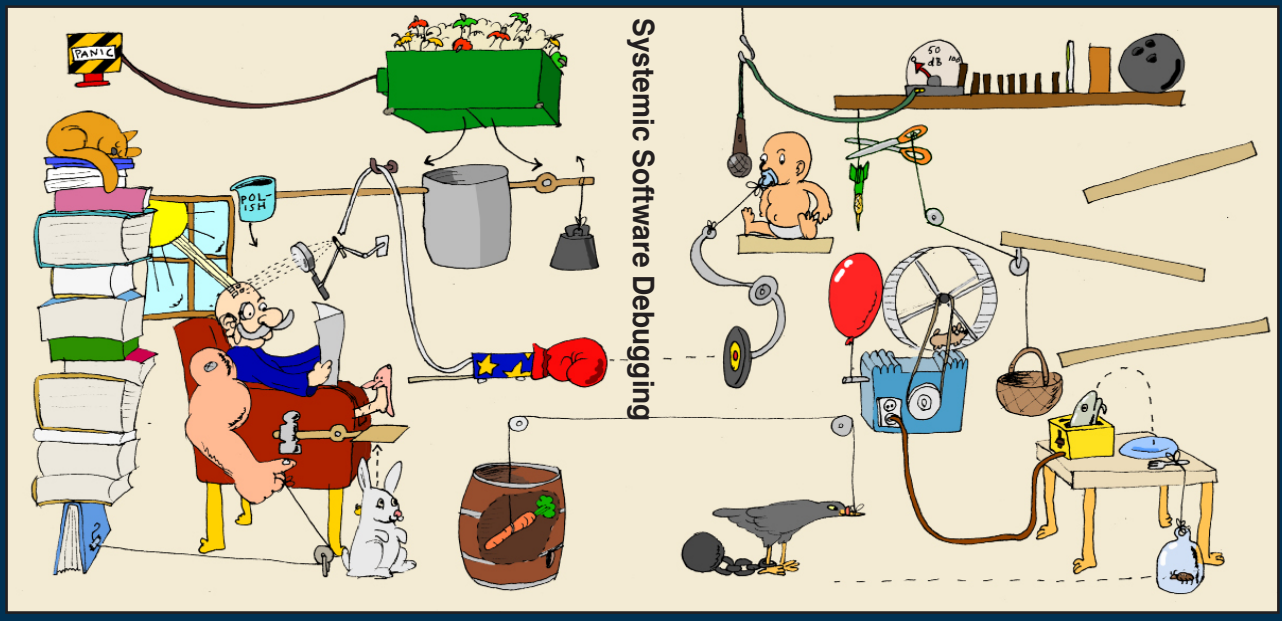

Systemic Software Debugging

Per Mellstrand & Björn Ståhl

Systemic Software Debugging

Per Mellstrand and Björn Ståhl

We dedicate this work to all ambitious requirement specification engineers and system architects who manage to keep up the good faith despite the fact that software produced with their work as input never ever functions as specified, intended or expected.

Authored by Per Mellstrand and Björn Ståhl.

v. 1.1

Contents

Acknowledgements

The ideas presented in this work have evolved over some time and thanks to input from many different persons.

We would like to acknowledge the valuable ideas and opinions we have received during discussions with colleagues at the MiB department at Sony Mobile Communications, formerly Sony Ericsson, especially input from Victor B, Björn D, Roger P, Christian P, and Max H. A special thanks goes out to Robert S for extensive feedback on everything from typography to method. We would further like to publicly thank and acknowledge the feedback from those that have participated in the 'Systemic Software Debugging' course throughout the years.

We would also like to acknowledge the support from Blekinge Institute of Technology (BTH), Sweden.

Finally, this is not a finished work but is in what the software world would consider *beta* state. *We greatly appreciate your input and look forward to an interesting discussion on software and software malfunction..*

To get in contact with the authors, please use the e-mail address: `contact@systemicsoftwaredebugging.com`

Preface

The views expressed herein are those of the principal authors and do not necessarily represent the views of, and should not be attributed to, Sony Mobile Communications.

This work was initially written in 2009-2010 for the use as supplementary reading material part of a seminar- oriented course in systemic debugging with a target audience of engineers and senior engineers at Sony Ericsson and as such, it has been edited somewhat to remove references to internal and/or confidential tools and systems. It is released publically and freely (CC-BY-3.0) in the hopes that it can be of some use to other curious minds around the globe.

People working with software in general and software malfunction in particular tend to be quite busy. Perhaps this is so because they waste too much time in long meetings, as their days are spent accepting party invitations, or simply because software-intensive systems tend to have a bit too much *mal* in their function. So, while we fully realize that you have a busy schedule, please bear with us for a few pages in which we introduce this book (and for many readers the course in systemic debugging) and present initial arguments for why reading this book (taking the course) is worthwhile and relevant for your work.

Software Construction, Execution and Failure

Modern software systems are complex, interconnected information-centric systems far from the algorithm implementations we figuratively call *punchcard software* constructed a few decades ago. These systems are developed iteratively by large groups of developers, and few people, if any, have a true whole-system view of how the system holds together, not to mention how it behaves during execution.

For these systems it is hard to give an understandable, unambiguous and objective definition of what constitutes a *bug* or *defect*. Some attempts have been made, ending up with explanations such as *"the developer adds a defect to the system"* or something similar which implies that there exists some objective truth as to how a system specified in an informal requirements specification should behave. Rather than tumbling down the way of unusable formalism, we simply consider a *bug* to be unwanted system behavior (according to some actor) with the typical restriction that it's non-trivial to explain why the system

behaves as it does. More often than not, bugs require fixing for technical, political, religious or other reasons. To keep the technical focus in this book, we do not consider the important topic of why bugs should be fixed but solely the technical aspects of how to explain hard-to-understand behavior in complex systems to enable fixing it for whatever reason.

Unbreaking Software

To fix, or unbreak, a software system one must understand what is *actually causing the unwanted behavior*. In practice this understanding means explaining the causal factors (what is actually happening) at an operationalizable level of abstraction (such that someone can use the explanation for fixing the system). In this book we use the term *analysis* to describe the process of finding such an explanation.

Analyzing a complex bug in a large interconnected system within a realistic timeframe requires extensive knowledge and a good set of tools and methods. Systems change rapidly, are deployed in heterogeneous environments (on different hardware and with different adjacent software) and always live longer than intended. This implies that preferably both knowledge and methods should be applicable in different environments and for different systems.

We address the difficult problem of software analysis from two perspectives: firstly, from a more philosophical one of working with complex systems where the causality often is hard to understand and where there are inherent limitations in what can be measured irrespective of the tools and methods used, and secondly, from a technical perspective where we exploit shared properties from programming languages, compilers and the machine to allow a generalizable approach to analyzing a wide class of systems.

Layout

The layout and intent behind subsequent chapters are as follows:

Introduction – This chapter briefly introduces the means and reasons for debugging and analyzing software and systems thereof together with some of the various problems that may be encountered along the way. In spite of its name, this is not to be regarded as an introductory chapter as such but rather as a glimpse into the field of debugging as a whole. This chapter in particular is purposely somewhat dense and abstract. Therefore, it is advised that you, *the reader*, alternate between this chapter and the ones of a more technical (Software Demystified, Tools of the Trade) nature.

Software Demystified – In this chapter, the major components behind software are described with focus on parts and processes often left in the dark or forgotten entirely. In essence the scope is that of *compile and run*. Thus we start at the

output of compilation and stop at the point where we have a process running our newly created executable. When doing so, we cover both the transformations performed on the description (program code) and the assembly line which makes it all happen.

Principal Debugging – This chapter builds upon the foundation and problems poised in the *Introduction* chapter and delves into the very specifics of applied debugging for a number of different targets. In this way, the chapter covers the entire span of exploitable generic properties that can be both measured and manipulated.

Tools of the Trade – This chapter covers the underlying mechanics behind the set of tools typically associated with debugging, tools that can be exploited to study and manipulate software in all of its various states. Bias is placed on problems and considerations often forgotten or hidden away behind the facade of development environments.

Both *Software Demystified* and *Tools of the Trade* are written in such a way that they can be read and understood independently of the other chapters. Because of this, there is some overlap and redundancy in terms of explanations if read in combination with *Introduction* and *Principal Debugging*.

Notes

On terminology – Debugging in the sense described herein emerged as a craft derived from the one of software development but has a considerably less enthusiastic following than that of its parent and has for the most part been a source of panic rather than inspiration, receiving comparatively much less attention in both writing and research. It is therefore not surprising that the technical terms in use vary between being overtly vague, ambiguous and redundant to riddled with misplaced analogies. The sentence 'a debugger debugged the bug using a debugger' suggests that there might be some work left in this regard and that the alphabet is, perhaps, not leveraged to its full potential.

Although the proper thing to do would be to provide a comprehensive list of terms to strictly follow from now and on, the chances that such a list would be adopted are slim to none. As there are matters of greater importance we will have to, for now, make do with the ones we have but slowly introduce alternatives through the course of this work and nit-pick whenever possible.

As a warm-up, let's look at some words that will be overused both here and in applied debugging situations.

Bug, debugging – That an entire subfield of computer science is named after the removal[1] of what amongst other things refers to insects, wire-tapping devices and a particular kind of car is unfortunate at best. Used synonymously

[1]Here de- is used in accordance with the English prefix for *getting rid of* or *removing*.

with defects and defect management, bugs and debugging are well-established but emotionally-charged parts of developer vocabulary (along with humorous and colloquial spin-offs).

Behavior – Saying that something behaves in a particular way describes the response an organism gives to certain stimulation. It is a far stretch to claim that executing software falls within the definition of an organism, at least in the biological sense of the word. Therefore, *behavior* has been broadened in software development contexts (and elsewhere) to describe a particular set of reactions (patterns of data and state transitions) to some stimulation or input (*code* and *data* or abstracted through *methods* and/or *function calls*).

System – A system denotes a collection of things with some assumed or implied common ancestry, shared role or property. It is furthermore a potential candidate for most frequently occurring noun in the history of academic and technical writing, worthy of being elevated to the status of grammatical conjunction.

Environment – The environment is what lies directly outside, and is connected to the current system. Consequently, the environment is what affects and what is being affected by the system but lies outside the model of the current system. For example, when considering the software part of an embedded device to be the system, the hardware of that embedded device constitutes the environment. Similarly, when considering only a single programming language function to be the system, the environment includes other functions that invoke or are invoked by the system, variables accessed and the executing machinery.

Virtualization – This refers to the act of adding an enclosing layer around some system in order to instrument and transform the system's interaction with its surroundings. Depending on which parts of the software or hardware it is that are being virtualized, the outcome may be called different things. A virtual machine is possibly the most ambiguous form of virtualization in that it can either describe the virtualization of an entire hardware platform from the perspective of executing software but also the bridge between a specific kind of mathematical model (an abstract machine) and a software or hardware platform.

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, messups, 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 somewhat extensive selection of synonyms, furthermore, seems to at least hint that either things go wrong more often than one would like or that we just tend to be overtly observant whenever things do go wrong. The somewhat frightening thing is perhaps that we 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 possess seem to outweigh most of the kinks and quirks they bring.

The ideal would, of course, be to somehow create software that is without snags or dents. 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 developers. They, in turn, sit down and with perfect rigor first develop the system, then deductively prove its correctness, to finally deploy the solution to a supposedly satisfied customer. In other words: the 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, these comparisons are both poor and misguiding at best. Software is not even remotely similar to a bridge, a chemical, a car, a quilt or some biological entity. In fact, the very characteristics that make software unique also constitute the reason why software is so difficult to model using traditional, mathematical notation. These characteristics spring from the strong interconnectedness between the software, the machine and their 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 band-width. In addition to this, software affects itself through feedback loops, concurrency and recursion.

When you look at phenomena that have characteristics that include interdependent information systems in close cooperation with a large and complex machinery, both the machine and the software are packed full of feedback loops affected by concurrency as well as by recursion and external influences, while at the same time working under constraints from limited and shared resources both in terms of computational capacity, storage space and communicational bandwidth.

An alternative to the supposed ideal of *flawless systems* would be to, with a systemic perspective in mind, *embed anomaly resilience*. This means that you design and develop systems with the notion that components are far from flawless and do decay with time. Things will eventually go awry and this should be acceptable and manageable to a certain degree. This means that we:

1. decouple components, especially critical ones, in order to isolate problematic parts and control cascading effects,
2. implement self-healing mechanisms, in order to recover from component failure.
3. recurrently strengthen the system through dynamic hardening and continuously validating behavior, since self-healing and monitoring gives feedback on component stability and data on errors as well as their impact,
4. monitor component conditions and state in order to refine hardening and provide feedback for future development efforts.

These ambitions are combined in order to create systems resilient in terms of bugs and vulnerabilities. Such an ideal, however, is not only infeasible by current standards but also limited in scope to systems with certain problems and properties.

We argue that the core of problems experienced, – the problems that these ideals try to come to terms with – are transitional in nature and that the meaning and role of software are moving from one of *software-as-punch-cards* to *software-intensive systems* or even as far as *socio-technical systems*. This implies that our tailored principal approach towards debugging said systems should also transition from one of *software debugging* to one of *systemic debugging*.

The core of this work is thus based on the fairly primitive notion that these large and complex systems are not – and for the foreseeable future will not be – perfect in the sense that they are free from bugs. Thus, we must acknowledge the presence of bugs and identify ways to effectively find and remove these, even from large systems that we do not fully comprehend the function of.

For the remainder of this chapter, we will focus on establishing a common ground regarding the more philosophical areas of what software systems consists of, how they are developed, where in this process bugs are introduced

and similar areas of interest. Those less philosophically inclined may safely proceed to the next chapter, *Software Demystified*.

1.1 Demarcation

The overall scope of software analysis in general and debugging in particular far exceeds what can reasonably be discussed here. This section will therefore be used to define the boundaries of that which will not be covered in further detail.

Our intended audience is first and foremost software developers or system operators with some extended knowledge of the details and inner mechanics of the particular systems they are working with and, more importantly, people with the ambition to refine said knowledge. Bearing this in mind, we will consider bugs and situations which – in spite of whatever requirement specification that happens to be in play – can rightly be considered undesirables.

1.1.1 Analysis Techniques

As we want to be able to understand and manipulate properties and behaviors of software systems in order to strengthen decision making, we start with an orientation of analysis approaches available and find out which, if any, is most suitable to fit these ends.

Static or offline analysis, is in short the category of tools and techniques that focus on some source of a software system, looking for constructions deemed dangerous or inappropriate. This *source* can be the compiled binary or object code, but most tools operate on the *source code* as written by the developer. Ideally, static analysis tools are a part of the toolchain used to transform source code into an executable binary and assist developers in finding possible bugs in their work.

Technically, static analysis placed as part of the build process tends to have access to information that gets distorted or removed completely when turned into a machine-manageable format. Such information is usually of a more abstract nature (code comments, design documents, revision control notes, etc.) but also hard to directly make use of without human assistance for the same reason; it can, however, still be considered a potential advantage for these kinds of techniques.

Static analysis can also be found as part of some formal development methods, in particular parts related to improving some supposed quality; examples include code, design, architectural and requirement reviews either made directly or through simulation.

Dynamic, whole-picture, or on-line analysis deal with instrumenting a system during execution rather than some component used in its construction. The principal advantage of a dynamic approach is that actual *state* in the system is considered and there is no need to hypothesize about possibly unwanted system states.

The main risks with dynamic techniques such as injecting probes and tapping dataflow are that this kind of operation may affect the system in ways that are hard to predict. Examples include corrupting state or affect timing in that delays are introduced to parts of the system. Because of this, there is always the possibility of running into so-called *Heisenberg effects*, which in this context means that what we observe might very well be the effects of our measurements rather than what we wanted to study.

In spite of these drawbacks, examining the executing system is still the most prominent way of catching the more complicated bugs and will probably remain so for the foreseeable future.

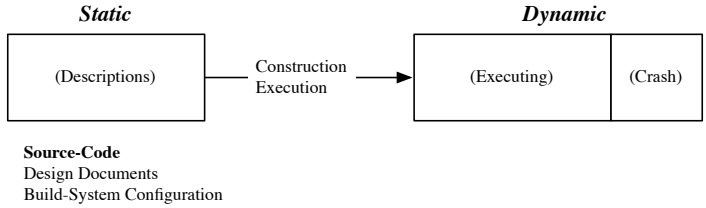

Figure 1.1: Analysis categories

Automated analysis – ignores the dichotomy between dynamic and static analysis methods, as it exploits properties of both. Certain kinds of profiling and testing fall into this category, as do some debugging approaches whereby you can describe what to measure, compare with, react to, etc in beforehand or at the same time as the actual instrumentation is performed on the executing software.

There are also more esoteric automated approaches to debugging as a whole that have the general idea that debugging is reducible to a state space search problem aimed at delimiting the supposedly sharp edge between correct and incorrect behavior. Prerequisites for this approach is a high repeatability rate and a formally specifiable expected outcome. The first prerequisite is due to the potentially high (hundreds of thousands) execution passes needed before the search space is exhausted, and the other prerequisite is needed in order to distinguish between a correct run and an erroneous one.

If it is both possible and reasonable to move a planned analysis session into the automated realm, it would of course be advisable to do so both in order to

optimize time by running the task as a parallel one to manual analysis but also for the benefit of testing for regression later on.

This categorization is by no means absolute, and the descriptions as presented here are mainly within the context of demarcation. There are of course tools that fit into all three categories, and borders can be further blurred through refined simulation models and high-level emulation.

In any case, *the focus for this and coming chapters is primarily on dynamic analysis as a means for grasping and refining an understanding of the particulars of a given system* and, to a lesser degree, how we can manipulate software during all stages of construction in order to ease future analysis.

1.1.2 When to Analyze

When talking about debugging and analysis, there are two entry points which will yield different results as the systems ending up being debugged and analyzed will barely resemble one another.

The first entry point is that of a developer working on developing software with typical characteristics of short cycles from the point where modification of some part occurs until any associated effects can be observed.

The second entry point is with a system that is by most standards *deployed*, which should imply that it is *stable*; many cycles of development, integration and testing have all been completed, the end results have already been passed to stakeholders, and it is now time to add improvements, adapt to new environments and fix problems.

Several key factors differ when these entry points are compared; during a rapidly changing, rapidly growing stage of development, various adverse effects and anomalous behaviors are tolerated to a different degree as they concern *software in development*. In terms of *coupling* (inter- and intra-dependencies), the software in development is loosely coupled to its environment[1] and its stakeholders.

When we are dealing with software that has been deployed (and possibly integrated into other systems), the scenario is quite different. With a supposedly stable system as per the previous definition, we are now dealing with a whole different beast altogether, one that can be thought of as tightly coupled in several settings, many possibly unknown. For any alterations made to the base that will be pushed to stakeholders (as a new release, patch, automated update, etc.) the cost, consequences and implementational complexity have all suddenly increased considerably.

A key attribute when estimating exactly how difficult it can be to establish causes for a bug, *repeatability*, may suddenly depend on factors outside developer control and influence. For the course of this book, we will assume

[1] This is not to be confused with coupling *within a model*, such as one made using *UML*.

a position at a sort of fictive border between the later stages of a software in development and a supposedly stable system where the majority of problems should be non-trivial to deal with.

1.1.3 Language and Paradigm

The ways in which we can articulate an envisioned software are many, but the productive ones are expressed in a form from which they, either directly or through some automated transformation, can be turned into instructions executable by a machine. The act of bridging the informal – our ideas, experiences and desires – to a productive formal representation is ultimately the task of a programmer.

Due in part to the size and complexity of this task, there have been lots of processes and tools developed, all supposed to assist in making things more manageable or in improving the end result in some way. The selection and configuration of processes and tools are the subject of development methods, and the stakeholders involved in such a process are software developers in the broadest sense of the word. Among the most influential – or powerful – abstract tools in this regard are programming languages and even more abstract programming paradigms, but what is important to realize here is that these are only tools to assist programmers, not the actual machines.

No matter which abstractions you rely on, it is ultimately the limitations and rules of the machine that will apply. The problems and approaches discussed herein will primarily be from the perspective of machines and the restrictions they imply, rather than singling out some specific abstractions.

1.1.4 Testing

On the topic of debugging versus testing, they are at once opposing, orthogonal and collaborating processes. Debugging in itself is mostly investigative and reactive by nature. The analyst gets feedback from somewhere regarding an issue that is to be examined, be it from other stakeholders directly or indirectly through some defect management tool. The analyst proceeds with the examination to the best of his abilities, ideally resulting in one less thing to worry about in terms of the project as well as another mark in the statistics.

Testing concerns verifying and validating behavior, through subprocesses such as unit tests, integration tests, model checking, and similar procedures. The fundamental issue in regard to debugging is – as Dijkstra so well put it – that *"testing can only show the presence of bugs, not their absence"*. For the analyst, this fact can be used as a tool, hence *collaboration*; if testing managed to show the presence of a particular issue once, it might just be possible to repeat this feat.

Repeatability is, as far as debugging is concerned, useful only when there is insufficient systemic data gathered in regard to an issue, meaning that the data

available is not consistent enough and/or not detailed enough to completely deduce the set of causes in play. Repeatability is also useful when a fix to a problem has been developed and deployed, a fix which in turn needs validation as software is necessarily tightly coupled at a microscopic level. This last point is unfortunately also where the unfriendly version of the classic game of *Whac-A-Mole* starts. A new fix or a different build means pretty bold and surprising changes in this tightly-coupled web of calculations, so the whole process begins anew.

Both testing as a whole and repeatability in particular stop being useful as tools when the information they might yield is entirely redundant or when in situations where they can have adverse and irreversible effects on the subject.

When testing is applied as a means to uphold some goal different than feedback to development or analysis, it is no longer a collaborating process but an *orthogonal* one.

Testing becomes *opposable* when there is competition for some shared resource such as a budget, manpower or access to the subject in question, as not all systems can be readily instanced just to perform tests. We will not dwell on the subject matter any further but will consider testing only as a tool among many others readily available to the analytic craftsman.

1.1.5 Software Security

Lastly, we come to the subject of debugging and security. If testing at times could be considered an opposing process to debugging, it would probably be justified to call security an opposing force rather than an opposing process.

Debugging relies on being able to isolate causal factors in play concerning a certain problem and removing them without otherwise causing additional harm to the particular system in question. In order to achieve this, a certain understanding of the system is in order, an understanding which can be achieved by analysis of said system. Analysis then concerns the study of system patterns – *behaviors* – where data comes from monitoring selected parts.

Although a large and prominent aspect of software security concerns the exploitation of vulnerabilities in software wherein the large majority of vulnerabilities rely on the same characteristics that would make us consider them *bugs*, there is little reason for us to make the distinction other than that not all bugs can be exploited to compromise security and are – in this context – therefore not vulnerabilities.

Another relevant part of security concerns the protection of system integrity and information. This is done in a variety of ways with the major categories being *obfuscation*, *copy protection* and *enforcing integrity*.

Obfuscation

Obfuscation is in direct conflict with analysis, as it is a counter activity focused on the concealment of meaning and has several uses. One use is to circumvent automated analysis by breaking up common patterns into alternate constructs, preferably ones that are harder to describe as patterns thus making them harder to find. This is possible because there are many ways to perform roughly the same systemic state transition.

A very simple low-level example of difficulties present when trying to establish pattern for static analysis use on an executable is the action of clearing a register on a `CPU`.

A common way of achieving this is by using an instruction for performing the `XOR` operation with the register as both source and destination.

A second way would be to simply set the value to zero with a `store` operation.

A third would be to have a series of calculations with the resulting value being the same as a reset register state and store the result of such a calculation in the desired register, and a fourth would be to have the value already stored somewhere in secondary memory and copy it over to the register and so on. There are literally thousands of ways to achieve this very simple operation.

While the different approaches mentioned are not equivalent, they are similar enough to be implemented with little risk for any adverse consequences (they tend to differ in storage size and execution time, if nothing else, but on a microscopic scale).

Another use for obfuscation is to hide the meaning of specific data, such as strings in part of the system, by applying some encryption or cipher scheme, effectively preventing static analysis. This forces the analyst to approach using more costly dynamic methods, which can be further thwarted by other techniques.

Even though the primary role of obfuscation is to conceal meaning, the secondary role is to act as a stall, making analysis more difficult and time-consuming, thus prolonging the time it takes to build an understanding of a system. However, with a successfully completed analysis it is also implied that obfuscation has failed in line with one mantra of security: *"security is not obscurity"*.

Copy Protection

Copy protection is applied to prevent unauthorized instancing of a system. It is mainly enforced dynamically, and there are two major ways to approach it.

One way is by preventing a copy to be made in the first place, usually by exploiting imperfections and ambiguities in mechanisms needed to construct the copy (reading source data in its entirety and writing said data to a medium

of comparable and compatible technology), which results in flawed copies, ones that are damaged enough to prevent or severely affect reusage.

The other way is by validating that the software is in some way unique by using hardware keys (often referred to as ”dongles”) or virtual keys with external validation such as serial numbers with a centralized server or call center for authentication.

Copy protection conflicts with experimentation and cloning as approaches for speeding up analysis through parallelization.

Integrity

The last category, *enforcing integrity*, is also a dynamic process typically found in the shape of a runtime check against modifications of both environment, and the software itself.

Checks against the environment often target debugging interfaces, virtual machines and external library dependencies whereas checks on the software itself come in the shape of checksums on data segments and on rare occasions, performance and behavior validations.

Copy protection mechanisms can all be circumvented but at considerable cost in terms of flexibility and time. Such circumventions also limits ways that probes can be constructed and inserted, therefore affecting measurements gathered.

Typically a combination of techniques from all three categories are employed to protect software from being tampered with. For examples on how complicated and involved analysis of such targets can become, please check the references at the end of this chapter.

For the scope of this work, we will assume that we are looking at systems where we already possess the means necessary for removing these kinds of protective measures from our subjects.

1.2 Software and Software-Intensive Systems

Terms such as software, system and software-intensive system have already been used several times during this introductory chapter and will be used throughout the rest of the work in a more or less interchangeable fashion. However, trying to dissect and discuss systemic debugging without some sort of insight into the somewhat treacherous and ambiguous terms used for describing various kinds of systems within this domain would be a somewhat futile endeavor.

Earlier on we hinted at an ongoing transition from *software-as-punch-cards* (figuratively speaking) to *software-intensive systems* which will now be given some additional detailing.

Punch cards as a way of storing information predates modern computers by far. They were initially used for storing instructions directing mechanical automation rather than computation. With the advent of early computers, punch cards were still an accepted and useful form of storing instructions for computation as other form were comparably costly and cumbersome.

Discarding punch cards as a means for storing instructions and data, we can assert that the kind of software created was still the automation of mathematical calculations where the computation was costly or complicated enough to warrant mechanical rather than human processing, for instance tasks such as the numerical approximation of non-linear differential equations.

While computational power increased, computers shrank in size, programming language capabilities grew and computing expanded from the execution of single algorithms targeting a specific problem to the blending of algorithms and calculations with a sort of structural glue, of which current OOP (Object-Oriented programming) can be considered an advanced form.

Somewhere along the same lines, computation found several new key areas of application. The managerial aspects of computational tasks took hold and virtually exploded in scope and reach; Operating Systems sporting features such as concurrency and process isolation with virtual protected memory. Secondary storage of instructions switched from punch cards to other forms of persistent storage, mainly hard drives. Programs moved away from being one-shot calculations towards becoming abstract tools for data manipulation.

Each and every time, complexity increased to seemingly unmanageable proportions, and new layers of abstraction were introduced and piled on top of previous ones, from machine code to assembly language to structured programming in imperative languages and onwards. Communication technology was introduced particularly in the shape of computer networks allowing for the collaboration between different types of machines, and new usage patterns and application areas opened up.

At this point software has turned into a dynamic entity composed to a large degree of building blocks shared with other programs, capable of moving between machines and reshaping itself as a reaction to external stimuli. We are now moving away from the realm of *software-as-punch-cards* and into *software-intensive systems*, where systems are composed of many different kinds of software, running on a variety of machines in close collaboration with other kinds of devices and processes both mechanical and human.

Even the tradition-bound distinction between software and hardware is significantly more blurred compared to just a few decades back. There is *firmware* hidden in one-chip-solutions, constituting embedded devices with capabilities exceeding that of traditional stationary computers. *Microcode* remaps the way instructions are executed on a processor. Shaders control graphic adapters, and so on.

1.2.1 SIS Example: SCADA

As an example of a software-intensive system rich in legacy, consider an instance of a SCADA (System Control and Data Acquisition) system, 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.

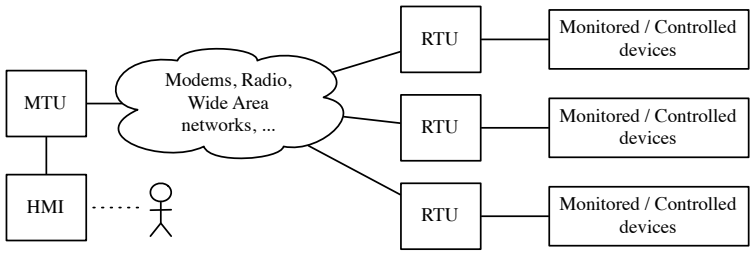

Figure 1.2: A Simplified SCADA Model

At this point the SCADA model (Figure 1.2) emerged, with RTUs (Remote Terminal Units) gathering measurements and enforcing commands. Each RTU communicates with a MTU (Master Terminal Unit) that aggregates and records data but also interacts with a HMI (Human-Machine Interface).

The HMI is used by one or several operators that manages 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 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 HMIs. 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, we need to take a step back and for a second just consider 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 possibly leading as far as to the Internet.

Surely this solution is, judging from history, anything but bug-free in any sense of the word. So, how many of these systems does anyone still employed comprehend? Enough to swiftly strike down the odd bug that appears? Enough to add features without fear of side effects such as broken dependencies? Will anyone know what to do if – or rather when – a serious problem occurs? Current abstractions, or adding new ones for that matter, will do little more than mislead while one is standing knee deep in cascade effects from some twenty year old low-level oddity.

1.2.2 Dealing with Complexity

Putting the previous example aside, some questions remain. How do we deal with complexity? How do we manage attaching new functionality and correcting certain flaws? How much and which parts of these systems do we understand?

One aspect of our dealings with these kinds of systems is that of a *black box*. To clarify, on an individual basis there are parts of most systems which are somewhat foreign in that we do not know about (or chose to ignore) some of the inner workings but instead we focus on details of its possible application or function. These *assumptions* along with one or several *interfaces* (sets of inputs and outputs) allow for a system to be used both directly as a tool towards some end and as part when creating other systems. In the latter case, where several systems connect through each others' interfaces we get *borders*. At these borders we can establish *protocols* as means for validating the connection and information exchange across systems, regardless of any assumptions in play.

Evidently, neither a black nor white box metaphor of reasoning is particularly suitable for depicting the way we think about complexity in software systems when trying to unwind causal sequences during a savage bug hunt. One does not simply attempt to senselessly map inputs and outputs without being close

to or fully engulfed in despair. Although it might not be formally described, we do have some hunch as to the inner workings of a particular black box either through the context of surrounding components or through necessities derived from its function.

Although there might be several ways to, for instance, save the contents of a video or memory buffer as a JPEG-encoded file, can it be achieved without reading said memory or without sending the contents through a discrete cosine transform? Even though we might box the software in this case as an image processor we, as stakeholders, do possess knowledge about some of its functionality and, through our profession, some of its implementational necessities. These are sufficient vantage points; as users we have some idea about what the software we are using actually does, and, as developers, how some parts must be made. What is missing is precision, which is fine as long as it is not needed, but as far as undesired system states go, *the thin line between a coherent system and an erroneous one can be crossed by the execution of a single instruction*.

The alternative then, which is about being securely aware in regard to inner mechanics of some part in question, means that we are either not dealing with complexity, making this entire point moot, or we are being naïve concerning some of the software characteristics previously discussed. For a given software system, merge together the amount of state transitions in play, the many levels of abstraction applied as well as the concurrent and distributed tasks being performed then try to faithfully answer this question: *do you know exactly what is going on?*

2003 NE American Blackout

To amplify the relevance of the discussion on software-intensive systems, complexity and the upcoming one on causality, we will now take a look at a real-world example of the extensive and bizarre consequences of flaws in software-intensive systems.

Electricity is no doubt a key dependency for most services and societal functions these days, and the power grids that electricity is distributed through are large and costly structures that span the width of entire continents. These grids are sensitive to disturbances and repeatedly fall victim to the forces of nature. Lightning, snow, storms and other heavy weather events along with wildlife and sabotage all contribute to small but local disruptions on a daily basis.

There are several safeguards added to the power grid to protect against, first and foremost, the acts of mother nature. One of them is through redundancies. There are several possible paths for power to travel in order to reach a certain destination. If one of these were to be disabled for whatever reason, another route between the production facilities to the end consumer would take its place. Some protection measures are added at the various borders between different components in this system, i.e., between power lines and

transformers, households and so on. If a measurable property such as load, current phase or frequency would deviate from a preset tolerance margin, the protective device would cut power and possibly notify the proper authorities through some alarm mechanism via the associated SCADA system. Already we have introduced a circular dependency; the power grid is controlled and governed by computerized information systems, software which in turn relies on electricity in order to function.

From time to time, but not very often, something happens which cannot easily be predicted. A series of coincidences manages to bring down one of the more important nodes in some region of the power grid and ultimately skews the distribution of load and supply across the grid at large. Somewhere this increased strain exceeds some preset threshold, acting as a trigger to the protective measures installed, bringing down additional nodes and so on. Suddenly, a large part of the grid is disabled and there is a so-called *blackout* in the power grid.

A large blackout took place on August 14th, 2003 in the northeastern region of the North American continent, leaving roughly 50 million people without electricity. The cascading effects from a comparatively trivial problem had escalated into something that paralyzed vital societal services with costs well into the range of billions of dollars. Not long after the initial blackout, speculations on the underlying cause were running rampant blaming everything from the `BLASTER` software worm to falling trees, incompetent management and poor design as cause of the event. We will not delve into the specifics here, but for further reference please consult any of the many reports on the matter, such as [NERC03][NYISO05].

Cited among the dominant initial causes behind the blackout was a software bug in one of the many pieces of code constituting the particular `SCADA` system. As the actions an operator can take in order to ascertain the well-being of the grid are heavily biased by the information about the current status of the governed system at large, it is vital that the information is both accurate and recent.

In an interview with a representative of the software manufacturer that uncovered and analyzed their end of the problem, some additional information on the specifics of the bug was provided [POULSEN04]. In one part of the system responsible for logging and managing alarm events, there were several concurrent threads of execution, a few of which shared reference to some data structure for storing these events. The flawed implementation did not consider the concurrency properly, and some sort of inconsistency occurred followed by degraded performance increasing response times from milliseconds into the range of minutes. The operator view that was fed with information from, amongst other things, the alarm events was suddenly outdated, effectively preventing necessary immediate action to be taken.

Bear in mind that this was a software-intensive system with over three million documented hours of stable operation with fairly complex properties and

demanding requirements with considerable developmental effort placed on testing and deployment.

Rhetorically speaking, why is it so hard for us to swiftly discover and remedy flaws causing catastrophes such as the one described?

1.3 Cause and/of Panic

Causality has been mentioned once already during the Demarcation/Security section, where we stated that *"debugging relies on being able to isolate causal factors in play concerning a certain problem and removing them without otherwise causing additional harm to the particular system in question"*. Unfortunately, the who, where and what of causality is a complex philosophical topic shrouded by numerous conflicting interpretations and although this is not the proper forum for a detailed discussion on the subject, there is some merit to bringing it up as it is considerably hard discussing debugging and analysis without some primitive idea of causality in place.

Most have probably played with dominoes, lining up tall bricks so that when one falls it touches the next causing it to fall down too. A brick falling over is an effect, or consequence, of some force being applied to it, exceeding the brick's inertia and thus tipping it over. In motion, the brick collides with the next brick in line, applying similar force (with gravitational help to combat loss from friction); this continues until the last brick has fallen.

Another device similar in principle to the dominoes is the so-called Rube Goldberg machine, named after the now deceased American cartoonist Reuben Goldberg (albeit the idea employed is not exactly novel judging by numerous expressions of similar contraptions used in other parts of the world, from *Storm P machines* in Denmark to *Pythagorean devices* in Japan). All in all these terms denote *absurdly-connected machines*, which perform some simple task in a very convoluted way, often with an intended comic effect.

If you have seen commercials, comics or other pieces of entertainment wherein a skateboard rolls down a wooden-board and knocks over a soup can causing the contents to be poured down a funnel which then fills a glass that will eventually weigh down a scale which when maxed out switches on a toaster that eventually pops up two pieces of warm toast for someone's breakfast, or something similar, you have encountered something like a Goldberg machine.

An interesting difference between these sorts of machines in comparison to dominoes when viewed as systems is the properties exploited. Dominoes typically share a common size and shape but all work using the same underlying mechanism. While some parameters (such as decorations, wear and tear, etc.) differ, dominoes as depicted here can be considered a homogeneous system. Rube Goldberg machines on the other hand use off-the-shelf components (whatever you can find the imagination to (mis)use) in a wide variety of ways; they can be considered as a heterogeneous system.

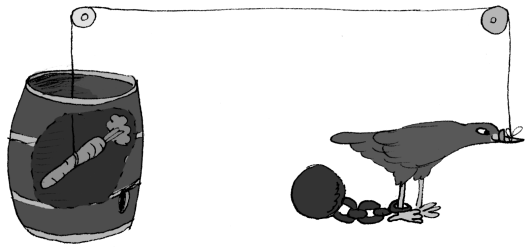

Figure 1.3: A partial Rube Goldberg machine

A Warm-up Example: Dominoes / Rube Goldberg machines

As an exercise, take a look at Figure 1.3. Can you, from this partial view of the system: a) know if the system is working, b) explain what the consequences would be from a component failure, and c) would you dare to change anything? Finally, take a look at the complete picture in Figure 1.4. Hopefully, the nature of the machine will be a bit clearer. However, looking only at source code, compiler output or some other small part of a software-intensive system shares the same fundamental problem with the partial system view: chances are that the component you have just singled out for study is used in an way that is very different from what you assume.

In a similar fashion, consider software code written in more esoteric languages or written using less widely-known constructs in otherwise popular structured programming languages[2]. With current software, chances are high that you will run into abstract Rube Goldberg machines as some part of a piece of software, but since we are mostly concerned with the final effect (like getting the glass filled with water) and to a certain degree lacking the ability to judge the merits of software we are oblivious to the insanity involved in the process.

Let us return once again to the homogeneous dominoes, where things start getting tricky. The (somewhat misleading) *domino effect*[3] refers to the chain of cause and effect that appears when you look at the first domino brick tipping over as the cause of the toppling of the next domino brick while still being

[2] An example would be the entries in the IOCCC – International Obfuscated C Code Contest, http://www.ioccc.org

[3] When such a series of effects is unexpected/unintended they are referred to as cascade effects rather than domino effects.

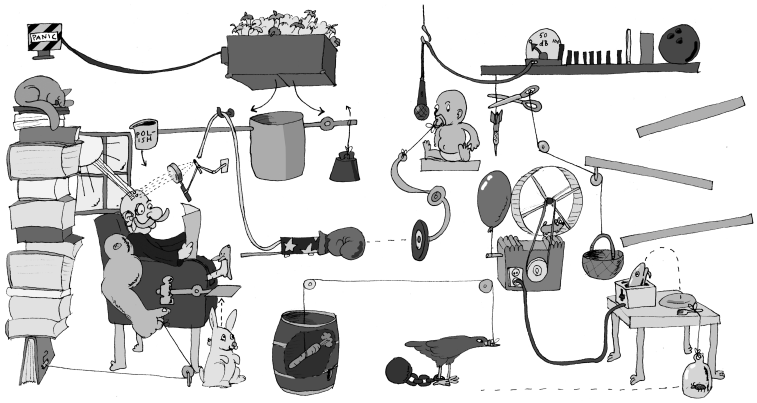

Figure 1.4: A complete Rube Goldberg machine

under the effect of the first force applied. But what was the cause of the first force? Someone pushing it? A gust of wind? Luckily enough, we can remove this problem through abstraction in our explanation by using something akin to *some force exceeding F Newton* as the initial action. We can even extend this with notational trickery to explain how our domino game can continue onwards to no end given an infinite amount of pieces and space or what the current state and shape of the system will be at a certain point in time for certain formations and so on. More importantly, we can control and make predictions within this system without the notion of causality in the first place.

Is it as simple for our Goldberg machines or our software-intensive systems? Unfortunately not. To be fair, the category itself is a bit more abstract and open than our dominoes, as there are few constraints except that there is supposed to be some end result from a series of components that are not usually interconnected in our particular fashion. The initial mechanism can be virtually anything from a sneeze to an alarm clock. Who sneezed? Who set the alarm? If we tie the initial mechanism to a particular person, what caused that person to start the machine? Suddenly this form of reasoning borders on the absurd even though answers become important when we need to assign blame for some accident or crime. *To ease this somewhat, we can study behavior in terms of proximal and distal causes*. The proximal cause for the Goldberg machine would be the trigger that starts the machinery. The distal or ultimate cause is whatever event initiated the trigger in the first place. This is just an overlay subdivision when looking for explanations while unwinding some undesired effect in systems we do not fully comprehend. However, that holds true for causality as well.

1.3.1 Determining Cause

A major part of software analysis and debugging is determining which factors involved caused a certain undesired effect to occur in order to feed this back into upcoming instances of the software, preferably by changing some offline artifact like source code. This is complicated due in part to the incredibly wide reach of state. Lots of components in a computer maintain and alter behavior based on some local or systemic state. At the lower levels we have the notion of `CPU registers`, flags and cache that are altered through the use of CPU instructions in accordance with some instruction set. This happens at a rate virtually incomprehensible to the human mind with millions of state transitions occurring every second. As there are finite places for storing and maintaining information related to a particular state, these locations will eventually be overwritten again and again by the state belonging to other tasks. In the event of some bug, it is highly likely that the state important for determining previous causes and effects is long gone by the time we discover the undesired effects of some bug.

1.3.2 Determining Consequence

Although we are usually forced to backward chain from an observed consequence to a series of causes and effects, this is not the be-all and end-all of debugging. While the major challenge tends to be finding the underlying cause, especially with corrupted state, we usually have to intervene with the cause as well.

As any modification to a software system risks having adverse effects, potentially ones worse than the original problem, there is also the responsibility to forward chain, by trying to estimate the consequence our intervention might have. This is a very system-specific task, even more so than finding the original cause.

Another situation might be that we have some effect deemed critical enough that we have to deal with the effects immediately, as the underlying cause might be too costly, tricky or, due to some other external pressure, unreasonable enough to figure out and fix. This calls for us to fix and mitigate effects rather than eliminate the cause, something done by adding resilience mechanisms. Such mechanisms will be discussed at length in subsequent chapters.

1.3.3 Multicausality

So far, causality as discussed has been linear. We have some underlying cause that leads to an effect which involves a change to some system property that some other part is dependent on which causes more effects and so on in a chain-reaction that eventually leads to some effect that is observable. That is when the analyst comes in to heed the cries of testers and users, saving the day. One little snag in all of this is concurrency.

Modern software systems are all to some extent concurrent; there are other processes sharing both resources and state which means that some tasks might be carried out in parallel. This is troublesome on its own when not accounted for, but the situation escalates when or if several seemingly independent processes share some factors that when combined will trigger an unmanageable situation.

Causality Concerns: the Überlingen Mid-Air Collision

Let us start with a real-world example of causality problems that arise when trying to unwind a situation where system state has been irreversibly changed, with several interacting processes all contributing to the final catastrophe: a software-intensive system with high stakes and many stakeholders, air traffic control.

In 2002 there was a serious mid-air collision between a Boeing B757-200 carriage plane bound for Brussels, Belgium from Bergamo, Italy and a Tupolev TU154M passenger plane bound for Barcelona, Spain from Moscow, Russia. The collision took place in the air space above the southern German town of Überlingen, resulting in the destruction of both aircrafts and their cargo and 71 fatalities.

Main actors involved:

- TCAS – Traffic Collision Avoidance System, installed on both aircrafts. Instructs on evasive action (ascend or descend) on detected risk of collision
- B757-200 pilot-crew (British/Canadian)
- TU154M pilot-crew (Russian)
- Air Traffic Controller (ATC) – Person monitoring the air space, instructing pilots on which actions to take and informing them of events taking place

Reduced course of events:

Timestamp	Actor	Event
21:21:50	B757	Enters air space, requests ascent to FL (Flight-Level) 360
21:30:11	Tupolev	Enters air space at FL360
21:34:42	TCAS (both)	Warns about path collision
21:34:49	ATC	Contacts Tupolev, orders descent to FL350
21:34:56	TCAS (both)	Tells Boeing to descend, tells Tupolev to ascend
21:35:03	ATC	Repeats order to descend to FL350, Tupolev acknowledges
21:35:10	TCAS (B757)	Orders increased descent
21:35:19	Boeing	Contacts ATC, tells of descent initiated on request of TCAS
21:35:24	TCAS (TU154M)	Orders increased ascent
21:35:32	B757 + TU154M	Collides in mid-air

(This timeline, comparable to a system event trace, was derived from Appendix 2 of the BFU Report. Neither the derivate nor the original depicts all of the contributing factors).

Main contributing factors:

- Air Traffic Controller – the ATC was simultaneously trying to operate two different control terminals and was the only person in the control center during the course of the event. In addition, during this time of day the area of responsibility of the controller was increased, expanding the load further to include additional aircrafts.
- Radar System – due to an ongoing system upgrade, the radar system was operating under reduced capacity, resulting in longer latencies and lower precision. The operator was not fully aware of this situation.
- Phone System – due to an ongoing system upgrade, the main telephone system was disabled and the backup system had failed several months earlier, unnoticed, due to a software error. This prevented both emergency transmission codes and the proper hand-over to occur.
- TCAS – Did not take into account the situation of when a recommendation is disobeyed and did not inform the air traffic controller about recommendations given.
- Policy – European pilots were trained to follow the TCAS in terms of conflict whereas Russian pilots were trained to take both controller and TCAS into account.

These factors can be further refined into sub factors highlighting many underlying technical and managerial problems, but suffice it to say that if any one

of the major factors had been removed the collision could have been avoided. The point of this little example is to illustrate that the problem of determining cause in concurrent systems is far from limited to multithreaded software and that small software problems can have serious consequences. For an in-depth view, please consult [NUN04, JOHNS04] listed in the reference section at the end of this chapter.

Instead of an executive summary explaining the entire cause effect chain, take one of the contributing factors or main actors and, for the sake of experiment, assume that it is the underlying cause. Then try to envision the scenario unfolding. For instance, according to protocol it is customary that a controller offloads part of the workload to another controller, either in the same building or at another air traffic control center. As the controller in this case was alone, he tried to contact another center nearby. Unfortunately, both the main and the backup lines were disabled. Therefore, the controller was forced to work under restrictions that prevented him from managing the planes properly. Which of the contributing factors can be explained as the cause of the collision?

Causality Concerns: Deadlock

This is another example of causality difficulties that, perhaps, feels a bit closer to home. Made famous by concurrency and college courses on real-time systems, many are today familiar with the dreaded *deadlock*. The premise of this issue is that two or more concurrent processes relying on the same resource can enter a state where none of the processes involved can make any computational progress because they are waiting on each other to finish some dependent task. The conditions needed for this to occur are:

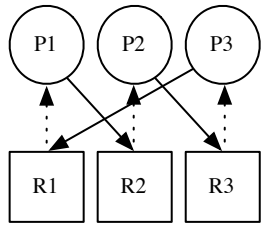

Figure 1.5: Deadlock example. The dotted line denotes a resource bound (exclusively) to some process, and the arrows denotes a resource the originating process is waiting for to become available.

1. Mutual Exclusion – The resource in question is assigned to a requesting process and can only belong to one process at a time.

2. Hold and Wait – The process with the resource assigned is waiting for some other resource to be assigned to it as well.
3. No Preemption – The process with the resource assigned cannot be forced to release it.
4. Circular Wait – The process with the resource assigned [a] is waiting for another process [b], which in turn is waiting for the process [a].

The problem is a bit more contrite than suggested above; the point here is not about giving a detailed explanation whys and wherefores of deadlocks and similar concurrency issues; for that there are far better explanations available elsewhere. The point we wish to make is, however, as this relates to causality, that none of these issues is direct cause of the deadlock problem. It is only when all four conditions are upheld that the deadlock effect appears. So we either aggregate and think of these four conditions as one cause to the effect and get the problem of connecting this to a particular part of the system (as they will be spread out and not found at some single statement in the source code or at some specific instruction) or we think of these as four distinct factors that just happened to coincide. In the latter case, simply preventing one of these from occurring solves the problem.

Computers are great tools for studying and experimenting with information systems, and many of the problems described herein are not strictly bound to the world of computers. For instance, compare the situation of deadlock just described to the *gridlock* problem depicted in Figure 1.6. Here we have a grid of one-way streets where the same conditions apply and with a similar outcome.

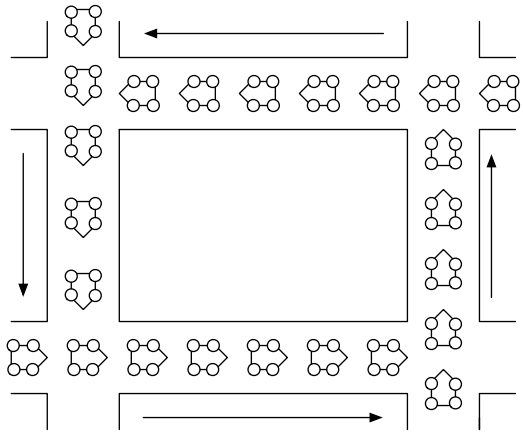

Figure 1.6: Gridlock example

At first glance, judging by illustrations of deadlocks such as *the dining philosopher problem*[DIJKSTRA] combined with the possibilities of combating each of the four necessary conditions, deadlock management and detection seem easy to fix and may even be possible to completely avoid during implementation. The reality, however, is that comprehending concurrency behaviors are far from trivial during development and design, with aid from development environments still in its infancy. Furthermore, concurrency is easily ignored when adding features to some old legacy-ridden monolith, especially when said monolith acts as part in larger Goldberg inspired software-intensive systems.

The moral of the Überlingen and deadlock examples, and the major overall point to this entire section, is that causality is both a philosophically tricky area and a relevant part of the challenges that software developers and analysts are forced to deal with, and finally, that often the starting point in a larger chain of events leading to some horrible crash will be set as the step where we gave up our hunt or at the point where we decided to put our collective foot down.

1.4 The Origin of Anomalies

An anomaly, or the deviation between *expected* or *intended* behavior and *actual* (or rather observed) behavior, is somewhat more complex than we initially depicted in the first paragraph of the introductory text; all other entries describe something unwanted rather than unexpected. By itself, an anomaly is neither positive nor negative but may be established as one or the other after evaluation. Anomalous behavior deemed as negative or undesirable is more in the vicinity of what is colloquially called a bug, but there seems to be some fine points to the classification. Consider the following example:

A developer is working on some particular feature on up and coming software, perhaps a few algorithms and a data structure or two. As the implementation moves towards completion, a test run with integration is initiated. At this point the developer has a fair but not precise idea of what the modifications are supposed to achieve. The modifications behave more or less as expected although one of the outputs just barely lands outside of a preset tolerance margin. Some minor fine-tuning is needed, as expected. Changes are made, and the solution in its entirety is committed to the project repository. Some time thereafter complaints start pouring in regarding problems with performance and stability. Retracing steps back and forth shows that the problems described appeared at the same revision the new feature was committed.

There were two anomalies present in this example, the first being the slightly deviating output of the new feature and the other being the newfound decrease in performance and stability. The first anomaly deviated from the intended behavior of the particular feature, and the second anomaly deviated from the expected behavior (the system as a whole was slower and less stable than usual). That much is clear. As for establishing causal factors, the deviating

output is already sufficiently isolated to a controllable subsystem. The other anomaly is more worrisome since the possible correlation to the new feature may very well be coincidental. Another reason might well be that the new feature and subsequent rebuild had the compiler and linker re-planning memory layout, making a previously layout dependent problem hit more sensitive system parts cascading onwards and eventually leading to the observed effects, falsely attributed to the new feature. A third problem, albeit absent from this example, is the situation when the anomaly affects the system in a positive manner. This is perhaps not as frequently observed, but it possess the same underlying challenge in that we still want to establish causal relations in play; this time, however, we want to take advantage of them rather than eliminate them altogether.

The principal claim on the origin of anomalies is that each and every bug is simply an *inconsideration on behalf of the developer*. As 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. Factor in other related concerns such as security, developing tests, proofs and wiggle room for future alterations into the mix and we are looking at a pretty complicated mess.

In order to be able to cope with such 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 best 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).

Furthermore, much of the transformation and execution is discarded when making considerations as these elements are considered either somewhat of a Black Art with only small gains at a considerable cost, or outside the needs and scope of the developer. This, we argue, is the most dangerous of inconsiderations in that each and every component in this development divide are also complex software systems, far from insignificant in regard to effect on system behavior with bugs of their own.

An early example of the dangers of overextended trust in the tools we use to construct software can be found in [THOMP84] wherein he 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 `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 one of these can you trust to not add subtle and hard to detect bugs and security problems?

The benefit to knowing about how these parts work (or rather, in what ways they do not work) and being able to distinguish between their respective problems – as long as you abstain from adding future complications into this potentially unsavory mix – is that one can be exploited to manipulate the state and behavior of the other.

A second claim on the origin of anomalies concerns the way we make *assumptions about the inner workings of some particular system* and how we thereafter proceed to make alterations for our own benefit by either changing the original source code or by exploiting some interoperability/modularity features (such as dynamically loadable libraries).

While any such alterations may work fine on their own when the subsystem in which they operate is considered, we may have inadvertently changed the situation for some other party that we indirectly share or compete with for some resource. If such a change was 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 bugs that may arise from such situations are complicated with behavior that may depend on environmental factors in the enabling machinery that can differ radically between each instantiated system. This happens both on a small scale with third-party library or as is the case with many modern development environments the built-in API where we assume and rely on tool and feedback from execution (like iteratively alternating between code completion/code insight for API parameters and a test run to see if the immediate behavior was as intended). This also occurs at a large scale from the integration of components and services when implementing, for instance, corporate wide information systems.

The third claim is the *ever-changing environment*, in 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 many times becomes a task even more complicated than developing the solution 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 are especially preventable; proving software correct requires proofs to be written with similar required considerations to be made,[4] so there is still room for inconsiderations and flawed assumptions. Also, the amount of independent software that works in isolation is shrinking, not the other way around.

1.4.1 Effects

If we combine the observations described and the arguments put forth in the previous sections, we get a fairly broad overview of how bugs come into existence, how to deal with this knowledge operationally, how far-reaching consequences a single bug can have on a larger system and finally, why the overlying problem with bugs has become so involved as it is. It is now time to make this view somewhat more precise by first refining the idea of bugs and secondly by revising claims on how such bugs can be squashed.

Not all bugs are the same, neither by underlying cause nor by consequence. This distinction is reflected by the terminology used colloquially when discussing bugs around the water cooler: *'race condition'*, *'buffer overflow'*, *'memory leak'*, *'deadlock'*, *'stack overflow'* and a wide variety of other, more colorful names. Some are very close to the cause that particular bug had, others the effect and many a combination of both. If these bugs lacked common properties worthy of discussion, for example if the effects of one bug could not lead to the creation of new ones or activate present but dormant bugs, then there would be little point in discussing effects by themselves. But as that is not the case here, we begin by taking a look at a simple classification scheme to ease upcoming discussions on principal debugging.

1.4.2 Tier 1 – Effects

As this is the first indication we get of the presence of a bug, it is also a convenient starting point.

As mentioned during the discussion on causation, we can reason about causes in terms of proximal causes and distal causes. In the same manner, we can attribute proximal effects and distal effects as means for subdividing a bug. In many cases some of the subdivisions can be ignored straight away; a terminal state is often a proximal effect with bugs in a non-resilient, tightly coupled system that will disappear once the distal cause is dealt with. By contrast, data corruption comes from a bug on its own but also hilights other ones that rely on very peculiar circumstances in order to be activated. For the latter case, although the effect may be dealt with simply by addressing the concurrency problem, the subsequent effects may deserve analysis of their own.

[4] What makes one formal notation better than the other?

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 to some distant device over a network connection, something else, often garbled, is written or the data itself is correct but is instead saved to an erroneous location/address[5]. 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 gets 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

Software systems contain an almost incomprehensible amount of state transitions executed in a machine with an environment containing safeguards against some states and illegal actions. When one such safeguards gets triggered, the initial response from the machine or environment is to redirect execution towards some pre-existing error handling code, usually a signal or exception handler. In lack of such a handler, the process or perhaps even all execution is terminated, basically putting the system in a dead or terminal state.

Inadequate Performance

Performance is a complicated beast, not only because it is a tricky property to reliably and repeatably measure during normal circumstances but because it concerns the main resource that all processes request and in some sense compete for, namely processing power. As if comparison between expected, intended and observed performance wasn't hard enough on its own, determining if we are actually looking at the first subtle hints of a killer bug or if we just happened to hit a scheduling/resource managerial dent is a tremendous challenge.

The only case that could be considered easy when looking at performance, comparatively speaking, is when execution grinds to a halt with little to no progress in computation.

In most programs there is considerable room for performance optimization at several levels with major decisions made through algorithm/structure choice

[5]If that location is in use by something else, it becomes an additional data corruption.

and by compiler voodoo. However, there are still to this day large gains to be found from hand-tuning at a microscopic level, unfortunately often at the price of source code complexity and at the risk of adding new bugs. In addition, all things performance (and to a similar extent, memory/storage) related are escalated when dealing with portable and embedded devices compared to more generic computers due to heavy constraints on energy consumption and, by extension, limitations in technology.

Side Notes

The categories selected for inclusion in this tier were based on an analysis standpoint using scenarios wherein one starts working from a report with complaints of some noticed effect. Intentionally absent from this list are direct conflicts with stakeholder requirements where the functionality implemented contrasts that of some agreed upon functionality.

1.4.3 Tier 2 – Label

With observable effects covered, we connect these to some of the bug names mentioned earlier. For each bug we will also consider some properties of its effect and potential problems concerning debugging and bug tracking. Properties in special consideration here are:

- Observer effect sensitivity/detection, or *Heisenberg effects* – denote the situation where the mere act of observing (or in the case of debugging, measuring) a system directly affects the state of said system[6].
- Repeatability/occurrence – called *reproducibility* by some; denote the estimated difficulty in reproducing the particular set of circumstances which revealed the effect in question.
- Impact/severity – a rough estimation of what kind of consequence the effect will have on dependent (sub) systems.

All these parameters rely, in one way or another, on various forms of qualified guesswork. When working with a large system covering literally thousands of versions, builds and targets spanning a large numbers of developers (into which some factor the amount of testers), however, chances are that decisions have to be made weighing in which discovered bugs have to be dealt with first and how much resources should be put into which kinds of related activity. Although we could deal with arriving bugs on a first-come-first-served order (or any other selection method) it is plausible that due to deadlines and similar

[6]This was briefly mentioned during the discussion on demarcation/analysis techniques.

constraints, we are forced to release a product with some (or lots of) already known bugs. In this grim scenario, the feedback to the product will be affected by (amongst other things) the amount of bugs, the frequency of their occurrence and the impact they have. A discolored icon in a user interface each third time the surrounding window is maximized surely deserves less attention than corrupted data being written to disk each and every time a user tries to save a document.

Let us take a look at some known labels and see how they fit into the picture described.

Race Condition

A race condition can rightly be called the mother of all concurrency problems and covers all kinds of situations where we have two or more concurrent tasks with some dependency on a shared state variable where one task can alter the state 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 hard to determine.

Dead- and Livelock

We've already looked at deadlock somewhat in Section 1.3. If we'd 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 (as previously mentioned; mutual exclusion, no preemption, hold-and-wait and circular dependency) whereas livelocks can occur simply from 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 another task but never is. In terms of detection, deadlocks are a lot easier than other kinds of concurrency problems, as the effect is immediate (several processes stops proceeding) and the impact is typically critical, with algorithmic approaches for automatic detection available. Livelock has a similar impact but is a bit trickier to detect because execution continues; it is just the instructions performed that do little to progress calculation.

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 re-

turning an error leads to an overwrite of some other data that happens to be stored in a location nearby; this fits the definition of data corruption effects pretty well. These problems are more apparent and dangerous in languages that operate close to the underlying machine. For a buffer overflow, the data structure is some sort of array[7] which is being indexed or addressed outside of its allocated space. Heap and stack overflows, on the other hand, describes not the structure being targeted but rather where the corruption takes place. A special case of buffer overflow is the off-by-one where an array incorrectly is assumed to be one element too large. This is common because in some languages array elements starts at zero, making n-1 the highest legal index value for an array of size n. Buffer overflows are reasonably easy to find and deal with when both targeted structure and supposed boundary are known, as there are external and dynamic ways of testing and enforcing boundary conditions. In addition, as repeatability is fairly high, they can quite reliably be manipulated to make their locality known and therefore making it easier to connect to where in the source code the problem originates. Unfortunately, buffer overflows occur quite frequently in software partly developed in languages that expose enough detail for them to appear, and 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 supposedly wrong (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. A double free is a specific such case in the C programming language where deallocation is performed twice on the same address and manifests itself in three different ways – as a crash, as data corruption or as a race condition:

- In order to manage the allocation and deallocation of memory, some 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 exception.
- If an allocation is made between the first and the second deallocation, chances are that the deallocation will affect memory currently in use by

[7] In C this is just a memory region.

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 having a very high likelihood of data corruption as effect.

- Lastly, some memory managers make little to no effort in trying to secure the data in its own control structures. The second deallocation request may then instead of a crash or exception lead to overwrites of data in such control structures, like 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 in systems which 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, but it is also particularly dangerous when aliasing occurs between functions because the programmer would then have to take into account how all functions manage memory.

Protocol / Type Mismatch

As mentioned in the beginning of Section 1.4, there exists a multitude of protocols that developers are supposed to follow in a variety of degrees. These protocols depend on an interface through which data is 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 exists 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.

Another example of type mismatch would be that of date/time representations. As the implementation of computers adds some kind of constraints

to the possible size of a particular representation, such as 32-bit memory addresses, there are large but finite boundaries to what these may represent by themselves. When attempting to convert data from one format to another with lesser boundaries, there may be a loss of precision. If this precision is neither acceptable nor accounted for, the situation is similar to that of a protocol mismatch.

A third example of a protocol that originates from the underlying machine is endianness. There are two commonly used ways to represent integers (16-bit and larger), known as little and big endian. Should an integer be stored to file on a machine using one type of endianess and later read back on a machine with the other, problems arise. The second machine will successfully read the integer but, because the endianess is different, it will interpret the data differently and consider the bytes to represent *a different value* than what was intended.

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 (RAM) to more abstract ones such as handles for threads, interprocess communication and 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 perform 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 bug outcomes.

Discussion

There are of course several effects missing in this list; some have been omitted because they do little to serve as anything else but indications of the presence of some bug rather than a particular one that we can distinguish from others. *Mandelbugs*, *Bohrbugs*, *Heisenbugs* and *Schrödingbugs* have all been omitted as they, despite being mildly humorous, signify practically nothing except that the developer in question is anything but certain of factors involved in a certain effect.

1.4.4 Omitted, Tier 3 – Causes

Had things been convenient and simple, we could take a test report describing a certain undesired effect and consult our debugging manual as to which possible category of causes that particular effect belongs to. We could then proceed to order causes by likelihood, and for each potential kind of cause list the source code locations where whatever conditions that constitute said cause exist and check them off one by one. We could make debugging methodical and perhaps even automated, if it was that simple.

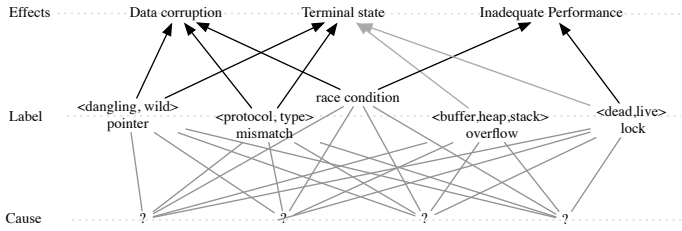

Figure 1.7: Cause-effect classification problem

Again, each anomaly can indeed be viewed as *an inconsideration on behalf of the developer*; there was some subtle requirement, some dependency or protocol that the developer failed to take into account when programming. The overall largest problem when debugging is not making changes to some part of the system to account for a bug, but rather identifying and locating all causes

involved. Anything that can help in this regard is obviously appreciated, but most effects can be quickly explained through *flawed or fault logic*[8], *issue with concurrency*[9] or *bad state transitions* [10]. However, none of them are useful distinctions.

This is where the craftsmanship in debugging takes over; there are lots of ways we can analyze a system's structure, information and behavior in order to deduce the causes of a specific problem, and we have a vast array of tools to speed things up. The craft, like in other professions, relies on selecting the right tools for the job, and when none exist, developing them.

1.5 Debugging Methodology

There is an ongoing discussion about how to formalize the work of debugging, or diagnosing, malfunction in software systems, possibly connected to the ever ongoing war on how software should be developed to meet some requirements considered having particular relevance. While having a structured approach is clearly to recommend over having no approach at all, the domain of debugging somehow seems hard to tame using a formal method.

A method, or simply instructions for getting something done, is in some cases a useful tool when trying to avoid repeating mistakes others have made in the past. Methods come in a variety of shapes at different levels of abstraction. Some methods for dealing with trivial problems, such as troubleshooting a washing machine, can be overtly complex and detailed. Other methods – perhaps for dealing with very complex problems – are too vague to be useful for any real-world problem. An example of such a method is illustrated in the following example:

1. Find the cause of the problem.
2. Fix it.

While this method is clearly meant as a joke and it may seem as it covers every possible problem in every possible domain, there are actually some problems that it does not address, for example situations where there is no clear connection between cause and effect.

A second method-related problem of particular relevance when working with software and other tightly integrated systems is the scope of the method compared to the scope of the system. When diagnosing a washing machine, one

[8] Software is little else than a carefully interwoven web of logic, saying that a bug can be explained with faulty logic is just a rhetorical tautology, neither interesting nor relevant.

[9] As an exercise for the reader, list all the concurrent activities that may affect the behavior of a software running on whatever desktop oriented operating system is currently in flavor.

[10] The only CPU instructions that do not explicitly alter state are either invalid or no-operations, which still arguably can be considered as altering state in the time domain.

considers only the function of the composite system but not the function of its components. For example, if the engine is broken, it can be replaced with a similar engine and there is little if any need to consider cascading effects into other components or systems. One can think of this as the method for diagnosing the machine (i.e., washing machine) is at a similar level of abstraction as the model of the machine. This is in stark contrast to diagnosing a software system where the inner workings of one component may severely affect the function of other components or the larger system. To diagnose such a system, one may have to apply different models (i.e., the system as a whole, system of components or even components of components, etc.), which makes it hard to use a method that implicitly assumes a single level of abstraction.

The third and final method-related problem of relevance is illustrated by how inexperienced troubleshooters regard detailed methods as a sort of recipe rather than guidance. When diagnosing a complex system, one will have to take feedback from the system into account, and there is no realistic way to do this work by simply following a list of instructions. A usable method for software diagnosis has to be applicable to real-world problems and take different levels of abstraction into account, but it cannot reduce the problem to a simple list of steps. It has to permit a flexible way of working because of the large scope of problems that must be tackled.

Take a look at another vague method:

1. Gather data[11].
2. Hypothesize an explanation.
3. Make a prediction.
4. Test this prediction.
5. If the prediction failed, repeat from Step #2.

This method suffers from several problems. Firstly, it says nothing about *how* data is to be collected (i.e., how to actually observe software behavior) but seems to leave this up to the investigator. This is also true for how to actually test the hypothesis on live software, which is something that can be very hard and require some method of its own.

Secondly, it seems to require that one can formulate a binary quantifiable hypothesis about the problem (i.e., a question which can be answered with either *true* or *false*). For complex problems in large software systems, this is an unrealistic requirement. It seems more suitable to allow different degrees of suspicion for different components (it is more likely that the problem lies in the parser than in the network stack, etc.) to allow for an iterative approach to finding the problem.

[11] This means observations on something that is unknown, unexpected or unexplained.

The third problem is that this is not really a method at all (at least not for debugging software), as it says nothing about the software or how to deal with problems in this domain. Rather, it seems to be some outline of a principal way of testing hypotheses, possibly borrowed from the scientific-deductive field.

Our point with this example and the discussion on debugging methods and methodologies is that there exists a plethora of guides and descriptions (often called methods) for software debugging. More often than not, there is some claim of scientific value in these methods, but given a closer examination the methods seem empty, irrelevant or trivial. In this book we lay out a number of principles on how to approach the difficult problem of debugging large software systems and how these principles can be combined to create a method. The purpose of these principals and the method is to provide some guidance on how to debug difficult software systems, not to imply a higher scientific value of the debugging work as such.

1.6 Concluding Remarks

We have now briefly covered a large area of software development and assorted topics. We know how bugs spring into place, what they consist of and to a lesser degree how they can be managed. We have seen how software systems have changed both in terms of meaning and scope from a small set of instructions representing a computation for a machine to carry out to the diverse behemoths that hide in plain sight in the shape of cellphones and similar devices. We have also been faced with examples of how initially good intentions can take terrifying turns when faced with the reality of the unexpected.

A major question still lingers however: *what is software?*

References

[BOYER04] Stuart A. Boyer, *Scada: Supervisory Control and Data Acquisition*, ISA International Society for measurement and Control 2004, ISBN 1556178778

[METZGER03] Robert Metzger, *Debugging by Thinking : A Multidisciplinary Approach*, Elsevier, 2003, ISBN 1555583075

[ZELLER06] Andreas Zeller, *Why Programs Fail: A Guide to Systematic Debugging*, Morgan Kaufmann, 2006, ISBN 1558608664

[TEL-HSE01] Matthew A. Telles, Yuan Hsieh, *The Science of Debugging* 1st Edition, Coriolis Group Books, 2001, ISBN 1576109178

[BLUNDEN03] Bill Blunden, *Software Exorcism: A Handbook for Debugging and Optimizing Legacy Code*, 2003, ISBN 1590592344

[COF74] E. G. Coffman, M. Elphick, A. Shoshani, *System Deadlocks*, ACM Computing Surveys (CSUR), v.3 n.2, p.67-78, June 1971

[THOMP84] Ken Thompson, *Reflections on Trusting Trust*, Communications of the ACM Vol 27#8, Aug 1984

[NUN04] *Identifying the factors that led to the Überlingen mid-air collision: implications for overall system safety*, Proceedings of the 48th Annual Chapter Meeting of the Human Factors and Ergonomics Society, September 20 - 24, 2004, New Orleans, LA, USA

[JOHNS04] Chris Johnson, *RFQ/137/04 Review of the BFUs Überlingen Accident Report*

[DIJKSTRA] Dijkstra, E. W., *Hierarchical ordering of sequential processes*, Acta Informatica 1(2): 115-13, Jun 1971

[NERC03] U.S-Canada Power System Outage Task Force *August 14th Blackout: Causes and Recommendations*, available at `http://www.nerc.com/docs/docs/blackout/ch4.pdf` and `http://www.nerc.com/docs/docs/blackout/ch5.pdf`

[NYISO05] New York Independent System Operator, *Blackout August 14, 2003 - Final Report*, available at `http://www.nyiso.com/public/webdocs/newsroom/press_releases/2005/blackout_rpt_final.pdf`

[POULSEN04] K. Poulsen, *Tracking the blackout bug*, available at
`http://www.securityfocus.com/news/8412`

2 Software Demystified

In this chapter we describe the principal parts – *the pieces* – of a software system and how these parts are put together – *the piecing* – into the puzzle we call an executable binary. In essence, these steps refer to the series of transformations made by a set of tools – *the toolchain* on one or several formal descriptions – *the source code* – which ultimately leads to machine instructions executing on a processor.

The principal motivation for this chapter and for the analyst to have a good understanding of software construction is twofold. Firstly, the analyst often works with reversing the chain of tools, e.g., from an executable where a bug manifests itself towards a high-level source where it seems suitable to fix. This is the opposite direction compared to how the chain of tools is designed to work, and working in this direction requires a good understanding of the transformation performed by each tool and how the tools are connected. Secondly, the transformation tools provide a viable source of information for the analyst, and it is important that the analyst knows how to exploit such information.

Layout of this Chapter

The layout of this chapter is as follows:

Firstly, we discuss the difficulties of predicting software behavior if one considers solely high-level sources, and then we argue for the importance of considering low-level aspects, the toolchain, runtime conditions and adjacent software when diagnosing a system. Next comes the main part of this chapter: a description of the transformation of software from high-level source to executable binary with the typical tools, formats and features relevant for analysis. After that, we look at how the executable is loaded into `RAM` by a loader and finally how the executing machinery enables the software to behave in a way an external actor can classify as wanted or unwanted. Finally, we discuss the cooperation between executing machinery and software and how properties of the machinery may cause unexpected and unwanted behavior to occur.

2.1 Hello, World. Is This a Bug?

To determine the behavior a particular piece of software will present when executing, one must have an extensive knowledge about the system at hand. This includes knowledge of the particular piece of software being inspected, as well as knowledge about adjacent software which constitutes the execution environment for the part being inspected. In addition to knowledge of the system as such, one must also understand how the high-level description is transformed into the machine-readable entity, which is where the bug actually manifests itself. As an exercise, study Figure 2.1 and try to envision possible unwanted behaviors the code will present when executed.

```
1  #include <stdio.h>
2
3  int main( int argc, char **argv ) {
4    char buf[50];
5
6    snprintf( buf, sizeof(buf), "%s␣says:␣Hello,␣world!\n",
7        argv[0] );
8    printf( buf );
9  }
```

Figure 2.1: Hello, buggy world!

When looking at this code, some issues – which may or may not be bugs – should be apparent to even a junior developer / analyst. At least the following four questions can reasonably be asked in regard to this source code:

1. *Will it compile?* Unfortunately, for many compilers and configurations this code actually compiles without even a warning. The problem which should prevent this code from compiling is the lack of a proper `return` statement in a function declared to return an integer. C is a well-defined programming language in which it is possible for the compiler to match the declaration and implementation of a function.

2. *Is* `argv` *used correctly?* To inexperienced developers, it may seem a bit strange to use a double pointer as an array, but in `C` that's perfectly fine. However, by using `argv` as is done on line 7 we make the implicit assumption that `argv` points to valid data and can be dereferenced (used). For example, if `argv` is `NULL` the behavior is undefined according to the C standard. In many environments dereferencing the `NULL` pointer causes the program to crash, but in other environments it may yield the actual data stored at address zero.

3. *Are the parameters to* `snprintf` *correct?* We have already made an assumption that `argv` is a valid pointer and that we can use its first argument.

However, we have not yet considered that there is a recipient of this data: `snprintf`. For example, it is not obvious how `snprintf` will react if the first element of `argv` is `NULL` (i.e., `argv != NULL`, `argv[0] == NULL`).

4. *Are the parameters to* `printf` *correct?* The use of `printf` in this example is suspicious at best. The first argument to `printf` is a string which may contain directives which are interpreted by the function and cause it to use one or more additional parameters. If there is a way to insert such formatting directives into `buf`, the behavior of `printf` may not be what the developer expected.

2.1.1 Finding the Answers

To find out if there is one or more bugs present in this small system, we have to dig into the four questions just asked, considering *the particular system*.

The question – *"will it compile?"* – is the easiest one to answer. Problems detected by the compiler can be tested for and almost always be addressed immediately when discovered. To answer the question as put, the only *environmental* aspect which we rely on is knowledge about the toolchain that transforms the source code into an executable binary. There is an invariant of this question which is a bit trickier to answer and can be put as: *if this code compiles, how will it behave when executed?* This is trickier because information about the toolchain as such is insufficient. We will return to this question shortly.

To answer the second and third questions, information about the source code and information about toolchain alone is not enough. We will also need information about the larger system and how it behaves during execution in regard to certain situations. This type of information – information about *the execution environment* – is essential for diagnosing many types of malfunction. To answer the second question, we must have information about how the surrounding system constructs arguments and transfers these to the program. In regard to the third question, the relevant piece of environment is the implementation of `snprintf` and how it reacts to formatting a string with a `NULL` pointer passed as argument to the `%s` directive. Typical ways of handling this situation are to crash or to insert some template string such as `"(null)"` for the `NULL` pointer.

The environment aspect of `snprintf` can easily be tested; write a test program which uses `NULL` as third parameter to the function and see what happens. The piece of environment needed to answer the second question is trickier to obtain, as there is no clear test case we can use to see if the value becomes `NULL`. The best we can do if we really must be sure is to test to set the value of `argv` to `NULL` and see what happens.

To answer the fourth question, we must have knowledge both of the environment and a multifunction scope, i.e., we must consider the execution flow in more than just a single function. Because the single input parameter to `printf` in turn depends on the result from another function, `snprintf`, its content must

first be examined in the context of `snprintf` in order to be able to deduce if `printf` will behave correctly. Fortunately, in this case there is some help available – a simple example which will cause havoc in the execution. Imagine that we rename the program `%n`. This means that when the program is executed, `argv[0]` will hold the string `%n` which will be copied into `buf` by `snprintf`. Thus, `buf` will hold the value `"%n says: Hello, world!\n"`. This value will cause a standards-compliant `printf` to store the number of characters so far printed into the variable pointed to by its next argument. Unfortunately, there are no additional arguments to the function, so this value will be written somewhere which is dependent on the machine. On an X86 PC computer this will almost certainly be the next element on the stack. As there is no such element in this example, it will actually write to the return address which means that when `main` returns it will try to resume execution at a new address (address zero).

Let us return to the first question – *"will it compile?"*. For the sake of reasoning, we will assume that the code actually did compile (it would if we use `gcc` with standard parameters), is used in some larger system, namely as a part in a software build system, and is executed by the well-known `make` program. Assume the code was targeted for the ARM architecture. The register `r0` is used then both for the first parameter and as the return value for a function. Without going further into ARM assembly, we can think of a `return` statement as moving the return value to register `r0` and then jumping to the address stored in the link register (i.e., a pointer to whatever called the function). This means that a function which fails to return a value will return whatever is stored in register `r0`. Typically this is the return value from the last function called, but it can also be some important data internal to the function.

In the particular case of our buggy "Hello, world", the function `main` is very likely to return the value returned to it by `printf`. The return value it receives from `printf` is the number of characters it printed on standard output. This value is dependent on the name of the program (as this is included in the output string). At this point the question is: what happens with the return value from `main`? Typically it is used as the error code for the program – something that the `make` program looks at to determine if it should halt further building operations.

2.1.2 Software Diagnosis and the Environment

Even for a tiny system, it is hard to tell exactly how the system will behave just by looking at a small piece of its source code. This is true both for human auditors and static analysis tools.

To answer the seemingly simple question of how many bugs are present in a less than ten lines long "hello world" program, we must consider the compiler (i.e., the toolchain), the environment (i.e., implementations of `printf` and `snprintf`) and possibly also the machine on which this system eventually will

execute. Some readers may argue that certain points we brought up are not valid, and some may point out more issues for their respective systems. Such arguments actually strengthen our point that you must consider the particular system at hand to determine what can eventually cause unexpected execution to occur.

To successfully diagnose your particular system, there are a vast amount of properties (limitations, incorrect implementations, etc.) of which you must be aware. For some systems one could try to make a list, but often you must be able to find out such properties as you diagnose the system. Many of these properties relate to either the way software is transformed into an executing entity or to the way different parts of a piece of software communicate during execution.

2.2 Transforming Source Code into an Executable Binary

A few years ago there were still simple software systems where a single source code file was either compiled or interpreted and constituted the entire software part of a larger system. For such a case, there is a straightforward connection between source code and the end output. For essentially all modern systems, there is a large set of tools involved in the transformation from human-readable high-level source code to the executable binary which a `CPU` can execute. This set of tools, called a *toolchain*, can be configured in different ways and contains different types of tools, but the principal way of constructing software systems follows a certain tradition.

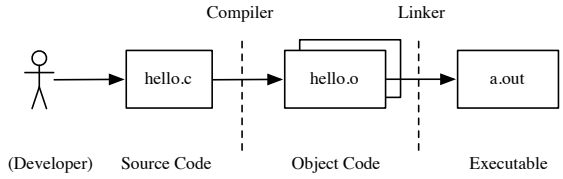

Figure 2.2: Hello source code, object code and executable

The key principle for a toolchain is that a series of tools is to be applied where the output from one tool is the input to the next. The initial input is what the developer has written, and the final output is something which can be loaded and executed by a `CPU`. This layout enables the flexible construction of toolchains where one component can be replaced without vastly affecting other tools and where a single tool can be developed with little regard to other parts of the chain. Figure 2.2 illustrates a simple toolchain with input, tools and output included.

2.3 Developer and Development of High-Level Code

It is the responsibility of the developer to write high-level code which is the initial input to the first tool in the toolchain, which in the case of Figure 2.2 is the compiler.

The actual job for a developer is to translate an informal description of a system in the form of a requirements specification, diagrams and documents into a formal system that, after being transformed into another formal representation, can be executed by a machine.

There is no automated way to transform an informal description of a system into a formal description, and to do this task a developer must make a number of assumptions, generalizations and simplifications. Many bugs in software originate from incorrect assumptions in this phase and for the successful diagnosis of a software system it is vital to understand how a developer reasoned when he or she wrote a specific piece of code.

2.4 Source Code and the Compiler

A source code file contains instructions represented in a high-level formal programming language. Source code is written directly by a developer (as opposed to being the output from another tool) and is transformed by a compiler to some form of object code. Programming languages have a syntax designed to be much easier for a human developer to understand than the binary representation used in an object code or executable file. Thus, we can think of the source code file as a formal representation of a small piece of software expressed in a form designed for humans – not computer software – to process.

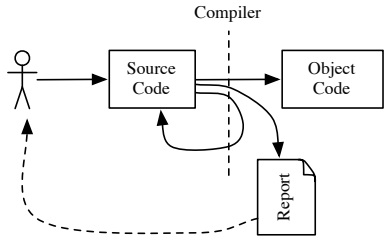

Figure 2.3: Compiler with input and outputs

There are some tool configurations other than the traditional human → source → compiler which operate on source code, as illustrated in Figure 2.3. The *preprocessor* is a tool which operates on source code processing certain special

directives and outputs source code which can either be handled by the compiler or another preprocessor. The preprocessor used with the `C` programming language has special properties and on a conceptual level is more or less integrated with the compiler (i.e., it is essentially impossible to write source code in C without relying on the preprocessor) while other languages have much more loosely coupled pre precessors.

Another type of static tool which commonly operates solely on source code is *static analysis software*. A static analysis tool operates like a compiler on a single source code file, but rather than producing an object file it produces a report with alleged bugs/defects. The main advantage for such a tool to operate on a source code level is that no other tool has been able to process the input and possibly shadow a defect. An apparent disadvantage is that the system still is in a form which is hard to analyze with good accuracy, typically making such tools either miss actual bugs (false negative), report nonexistent bugs (false positive) or both.

2.4.1 Interfaces

A programming language is a formal language (as opposed to a natural language such as Swedish) which implies that for a source code file to be *valid* for a particular language a number of formal requirements must be met. These requirements are often divided into two categories: *syntactical* and *semantical*. In this model a syntactical error is something that prevents the compiler from parsing the source code into compilable tokens and a semantical error is something that makes the parser produce a stream of tokens that doesn't make sense or violates language restrictions. A programming language places more restrictions on source code than merely syntactical and semantical correctness. Two such restrictions inherent in all high-level programming languages is the limitation of which entities may communicate and the nature of this communication. All modern languages are block-based, which means that the contents of a source code file can be modeled as a sequence of blocks where each block may contain zero or more child blocks. In languages based on `C`, the open and closed broken bracket (`{` and `}`) characters mark the beginning and the end of most blocks. The language places strict limitations on how blocks may communicate. For example, it is not possible to invoke (call) a block inside another function or access variables in such a block. The language provides a very limited set of communication primitives which is the only way these structured entities modeled as blocks may communicate.

An important consequence of this language design is that a programming language places a constraint on *program structure*. For programs written in the `C` programming language, the primary means of communication is function calls. Similarly for programs written in `Java` or `Ruby`, the structure is by requirement object-oriented, and the primary means of communication is method invocation. For programs written in `C++`, communication can be either function or

method invocation. *No high-level language allows for unrestricted communication between its principal entities or communication as free as the actual machine code allows. Understanding the restrictions placed on high-level languages is essential when hijacking execution, which is a key to tracing and debugging complex software systems.*

2.4.2 Local, Imported and Exported Entities

A compiler operates on one single source code file at a time, and it cannot consider arbitrary relations between different source code files[1]. While this places some restrictions on diagnostic messages produced and optimizations a compiler can make, it has two large advantages: it enables simple parallel compilation, and it requires metadata to be present throughout the entire building chain. The metadata is not only useful for the tools included in the chain but is also a very vital source of information when diagnosing the system.

To enable compilation on a file-by-file basis, the compiler handles three types of entities:

- *Imported.* An entity (such as a function, method or variable) that is used, but not defined, somewhere in the local file. There must be some formalized way of determining how this entity can be used or a compiler would have to consider all files as a whole.
- *Exported.* An entity that can be used (such as called or written to) by code from some other source code file. These entities must be created in such a way that the compiler can determine how to handle them when compiling the *other* file.
- *Local.* Entities defined in the local file and that are not visible outside that file. The compiler has complete information about how such an entity is defined and used, and it can do more or less whatever it likes to it as long as no external interface (internal or external entity) is affected. This includes inline functions, local function variables and – in many languages – private or protected methods.

The term *symbol* is used to describe these internal, external or local "entities" without discriminating between the type of underlying object the entity refers to. In most environments a symbol has a very limited set of properties, typically a name, a size and a type. For an example of different types of symbols, please consider the source code in Figure 2.4. The *imported symbols* in this example are the variable `version` and the functions `snprintf` and `printf`. There is only one *exported symbol*, namely the function `main`. The *local symbols* in this example are `get_major`, `get_minor`, `get_vstring` and the local variable `buf`.

[1] This is also true for languages where the compiler automatically analyzes imported files such as Java. Even though the compiler analyzes these files, this is done on a shallow level comparable to header files for the compiler to determine interfaces. It would be far too complex to consider the entire system at once when compiling.

```
1  #include <stdio.h>
2  extern short version;
3
4  inline int get_major(void) {
5    return (version & 0xff00) >> 8;
6  }
7
8  inline int get_minor(void) {
9    return version & 0xff;
10 }
11
12 static void get_vstring( char *dest, int szdest ) {
13   snprintf( dest, szdest, "hello, world version %d.%d",
14       get_major(), get_minor() );
15 }
16
17 int main( int argc, char *argv[] ) {
18   char buf[100];
19   get_vstring( buf, sizeof(buf) );
20
21   printf( "%s\n", buf );
22   return 0;
23 }
```

Figure 2.4: The symbolic language of programming

This example also illustrates that identifying symbols can be a little bit trickier than it first appears. On line 19 in the function `main`, there is a reference to `sizeof`. Despite looking like a function call, it is actually an operator in the `C` language. Other constructs that may look like references to external symbols can occur when using a preprocessor. For example, consider the consequence of replacing lines 4-6 (which define the function `get_major`) with a single line reading `#define get_major() ((version & 0xff00)>>8)`. In this case the preprocessor would replace the call to `get_major` on line 14 with its definition but keep the call to `get_minor`. Thus, despite two identical constructions, one is a reference to a symbol and the other is a macro invocation which is substituted before the compiler even sees it.

2.4.3 Application Binary Interface and Calling Conventions

When diagnosing software malfunction, it is important to understand the actual low-level function of a software system. As we noted earlier a compiler operates on a file-by-file basis, and the language it compiles always has a number of inherent limitations in how different entities may communicate.

When a compiler finds a reference to an *imported symbol*, it must generate code that can use this symbol and, similarly, when the compiler finds symbols which should be exported it must create them in such a way that they later can be

used from other files. This must be done *despite the fact that the compiler cannot have any information about the definition of these symbols*. In some cases one compiler may have generated a file which contains a symbol that is used from another file compiled with a different compiler. To handle these dependencies all larger platforms define a standard for how symbols should be stored and referenced. For data symbols (essentially variables) this is defined in something called the *application binary interface*, or *ABI*. For executable entities (functions and methods) the common name for this standard is a *calling convention*, of which there may be several that apply.

The binary interface for variables is typically rather simple. It defines properties and constraints such as the size for particular type and if there are any restrictions on how an entity should be placed in memory. This is in contrast to the *calling convention* which in addition to these aspects also specifies how parameters are transferred to a function and how the return value is transferred back to the caller. If there is any ambiguity in this interface, there is a clear risk that calling one function damages data for another. In other words, this may make the compiler generate a bug, an event that will not be visible in the source code of either the caller or the callee.

Another important aspect of the calling convention is that the compiler is forced to generate code that conforms to a specific interface which makes it possible – even easy – to hijack the execution at that point and do tracing of the system to find bugs.

2.5 Object Code and the Linker

An object code file, or object file for short, is the output from a compiler, and it contains compiled versions of whatever was defined in the file processed by the compiler. Although an object file contains compiled functions (i.e., machine code), it cannot usually be loaded and executed by a machine.[2] This is because the object file contains references to functions which were not available to the compiler (imported symbols) and often other loose ends which prevent a machine from executing the code as is.

There is a large structural difference between a source code file and an object file; the source code is primarily designed to be easily comprehended by a human developer and contains many different types of information such as instructions, constants and comments mixed together in logical entities such as functions. The primary user of an object file is the linker and eventually the machine which calls for a different organization of the information, namely one where each type of information is handled separately. While comments and other high-level markers are removed by the compiler, other parts of the

[2]There are some exceptions with certain virtual machines which allow the loading of an object file. This is possible because the virtual machine contains a small runtime linker that prepares the object file in memory prior to executing. There are no such features on a physical machine (CPU).

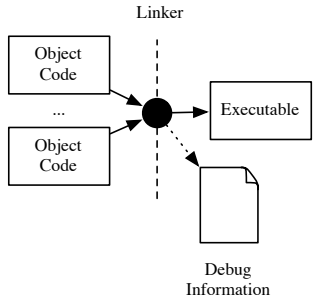

Figure 2.5: Linker; inputs and outputs

source code file are divided into logical blocks such that all instructions are placed in one logical part and all constants placed in a different one. These blocks are referred to as *sections* which is the principal logical entity used by the linker. Some sections do not contain any actual data but act as placeholders for the loader to fill with a prespecified byte pattern. Some commonly used sections are listed in Table 2.1 and, where applicable, the de facto name for such a section are also shown.

Common Name	Description
`text`	Machine code instructions
`data`	Data used to initialize variables and constants
`bss`[a]	Pseudo-segment for zero-initialized variables
`rodata`	Read-only data used to initialize variables and constants
	Debugging information for the respective source code file

[a] Acronym for the largely outdated term 'Block Started by Symbol'. While the acronym is widely used, the original meaning no longer has any relevance.

Table 2.1: Section Names and Contents

The linker combines one or more object files into a single output file and prepares that file so it can be loaded and executed by a machine. Principally, the only input to a linker is a list of object files, and the only output is the final executable file. In reality, for many systems a large amount of configuration must be given to the linker either in the form of command line parameters or a configuration file, and more often than not the linker produces some form of database suitable for debugging the final executable file in addition to the executable itself.

Much of the work a linker performs lies in matching exported symbols from one object file with imported symbols from others, making sure that all symbols which are imported actually exist. In some cases a linker may remove symbols which are defined (exported) but never used (imported). The linking task involves patching the machine code originally generated by the compiler so that the entire system holds together. From this description, it is actually the linker that compiles objects into a compilation – an executable – and a compiler translates between formal languages.

2.5.1 Principal Function of the Linker

The exact communication between the compiler and linker is highly dependent on the tool chain configuration and is somewhat out of the scope of this work. Also, when diagnosing a system, the most relevant parts of the linker are actually in the secondary inputs and outputs. However, in order to understand some strange execution situations and find hard to catch bugs, a basic understanding of the main function of a linker may come in handy.

When compiling a file, there is information which the compiler cannot access, such as the definitions of functions from other parts, as it processes different parts of the system separately. To avoid a situation where the compiler would have to consider the system in its entirety (regardless of its size) when compiling each and every file, the way symbols are handled depends on their specific type. Whenever a reference is made to an external symbol (a symbol imported into the local file), the compiler reserves space in the generated stream of machine code so that when the symbol is resolved the linker has space reserved to patch the machine code with the final address.

When the linker combines the contents of different object files, it combines them so that all machine code is placed adjacently at one address range and all constant data is placed adjacently at a different address range for each type of data it handles. It is at this stage that each symbol get its final address. There is no way for the compiler to predict where the linker eventually will place a particular symbol and therefore the compiler cannot generate the code required to glue a piece of machine code together with its data. As the linker has just torn apart what the compiler generated and reassembled the parts where it seemed appropriate, each and every reference from the machine code to data (typically only to large data which must be moved around) has to be patched.

Also, many processors have restrictions on how an instruction can load data which are dependent on the address of the instruction. This is sometimes referred to as that a piece of code not being address independent, as moving the code to a different address and executing it there will result in incorrect behavior. Since the linker has moved around the code, this too must be checked to assert that the machine code stream actually corresponds to what the compiler generated.

Finally there are further restrictions which severely complicate the linking. Some functions, such as interrupt handlers, must be placed at specific addresses and this is typically specified by the developer in the source code. However, since the compiler cannot control when it comes to which function is being placed at which address, it must forward such information to the linker. Also, because of strange and complex performance optimizations, the compiler can place requests to the linker on how the code should be placed in memory. Similarly, there is code which can operate regardless of where it is placed in memory. Such code is called position independent, or PIC (position independent code) for short.

The final obstacle for the linker is the restrictions (size, instruction set, etc.) imposed by the machine. For embedded targets, there is sometimes no way to combine the requirements from the compiler with size and instruction set limitations which originate from the machine. In these cases the linker fails, typically with some very cryptic error message.

2.5.2 The Linker Sees All

In a modern tool chain the linker is the only tool that considers the entire system at once. The compiler considers only one single source code file at a time, possibly together with some header files, but never the entire system. Similarly, a loader considers only parts that actually are loaded and ignores secondary information such as debugging information that may be embedded into a system. The principal function for a linker – to bind the software together – requires that the entire system is considered as a single large chunk.

It is important that one understands this fundamental difference between the linker and other components in a building tool chain when diagnosing the system. This is so both to understand *what can be done with the linker* and perhaps more importantly *what cannot or should not be done with a compiler or preprocessor.*

2.5.3 Modifying System Wide Properties

To analyze or modify global properties, i.e., properties which relate to use of imported and/or exported symbols, always first consider using the linker. Because the linker considers the entire system at once, there are always good provisions for analyzing and modifying such properties at this stage. Examples of global properties which can easily be analyzed/affected by means of a linker include:

- *How the invocation of a non-local function is handled.* For example, if you need to inject a special debugging/tracing version of a function, the linker typically provides good means to do so.

- *References to global data.* To get a list of users of a particular symbol (even if that symbol is a large array or similar), the linker always processes such information.

- *Uses of unsafe/deprecated functions.* The linker always maintains a list of cross references between functions. To determine if certain unwanted functions such as inherently insecure or deprecated ones are used, this can be deduced from the linker.

While some configurations allows an investigator to achieve similar results by using tools earlier in the chain, one should proceed with extreme care. For example, the `C` preprocessor can be used to substitute function calls. While it can only operate on a single file at a time, combining abuse of the preprocessor with a building system at first may seem like a good solution for tracing system behavior. However, transforming the code at this early level may cause several unexpected and unwanted side effects. For example, the new system may get other compiler optimizations, a different calling convention and a new code layout compared to the original. In contrast, operating at the linker level cannot affect compiler optimization or calling convention and provides a clean cut in the software for the investigator to measure or modify.

2.5.4 Trampolines and Stub Functions

In some situations the linker (or even the compiler) generates small functions that are only used to jump from one place to another. Such functions are known as *trampoline functions* or simply *trampolines*, as jumping there will only result in a second jump to the right place. Occasionally a trampoline function changes some aspect of the environment (i.e, modifies the calling convention), but in many cases the environment is left untouched with exception of the jump.

Similarly, other components in the toolchain can emit tiny functions which fulfill some specialized smaller task. All such functions, including the trampolines, are sometimes called *stub functions*, as they show up as function-like symbols but have a very limited functionality.

Stub functions in general, and trampolines in particular, are often used to circumvent limitations in the underlying machine or to explicitly assist in debugging. Some `RISC` processors, such as the `ARM`, place restrictions on how far a direct jump can reach (i.e., a jump encoded directly into an instruction) which results in compilers emitting many trampoline functions. From a debugging point of view, a stub function provides a good place for a breakpoint which is particularly relevant if the real function is placed in read-only memory or for some other reason cannot be patched.

2.5.5 Name Mangling

C and, more importantly, its ideas on how execution should be managed play a large role as structural glue between components of many different programming languages and is as such a de facto standard in software interoperability. While it may not be the best possible solution for piecing software of many different paradigms together, it has still proven sufficient for most known cases. Due in part to this role, the tools for other programming languages put a lot of effort into maintaining compatibility, with some conflicts as direct consequence.

One such area of conflict between C and other programming languages has been that of *overloaded functions*, a form of *polymorphism* which opens up a world of trouble when trying to interoperate across an interface like the one that linking provides. The idea behind function overloading is that a function may have several implementations that share the same symbolic name within the confines of the programming language. Which one of these implementations that ultimately is invoked during execution depends entirely on the amount of arguments passed and possibly on the data type of each argument. Now, the definitions of the large majority of type systems in languages such as C++ are only ever accessible to the static tools working on source code and are removed entirely during compilation. This reveals the problem when directly mapping the symbolic name of some exported function to one in the resulting objects, leading to collisions during linking.

```
1  fileHandle openFile(const char* filename, int mode) {
2    /* code goes here */
3  }
4
5  fileHandle openFile(const char* filename){
6    return openFile(filename, fileHandle::RDWR);
7  }
8
9  void fetchDrink() {
10   fileHandle handleBar = openFile("drink.pub");
11 }
```

Figure 2.6: Function overloading requiring name mangling

Consider the C++ example in Figure 2.6 where two different functions share the same name and one is dependent on the other. There are several ways to solve the collision problem, the dominant one being *name mangling* or *name decoration*. Simply put, it is a way of encoding meta data about some specific properties of a symbol into the name of the symbol itself according to some agreed-upon scheme. Instead of a symbol named `openFile`, there would be two symbols exported as `?zCOpenFile` and `?zCIOpenFile` or something to that effect.

Name mangling is used for other purposes than merely supporting overloading, for example in languages where some, but not all, functions can safely be called from C. Consider the example in C++ illustrated in Figure 2.7. The function foo can be called directly from C but bar cannot as its interface is dependent on language features unavailable in the C language.

```
1  struct parameter {
2    void print_it() {
3      std::cout << "Param:␣" << _param << std::endl;
4    }
5    int _param;
6  };
7
8  void foo( int param ) {
9    std::cout << "Param:" << param << std::endl;
10 }
11
12 void bar( parameter &p ) {
13   p.print_it();
14 }
```

Figure 2.7: Use of language features requiring name mangling

By using name mangling the compiler creates symbols (exported as well as imported) which the linker can handle without any special considerations. These symbols, however, have names that make them impossible to call from another high-level language[3]. Thus, C, C++ and functions defined in other languages can safely exist in the same executable, be linked by the same linker, and yet adhere to different rules imposed by the respective programming language.

2.6 Executable Binary and Loading

The executable binary file is the principal output from the linker and the system in its final, static, form. The linker has translated the executable binary as far as possible meaning that symbol names and references have been replaced with addresses which can be handled by a CPU.

Despite being translated to a format close to the machine, an executable file cannot be executed as is. For a CPU to actually execute the file, it must be directly addressable in memory. The process of making an executable file *actually* executable for a CPU is called *loading* the file.

In a traditional system, such as an embedded system or early UNIX-like systems, the process of loading an executable is fairly straightforward. The executable file contains a small header which describes the size and base address

[3]In this example this is so because the names start with a question mark.

for each of its sections. Each section is copied to memory at a given address. Some parts of an executing binary, such as space reserved for a stack and a heap, are filled with a simple byte pattern when the program starts and for this reason are absent in the executable. The size and expected address for these segments are still present in the executable file, and the loader creates them in RAM as if they were actually present in the executable file. When all sections of the executable file are loaded, the loader transfers execution to the start routine of the system. The address of the start routine is either well-known (always the same address) or specified in the executable binary file. In a modern system the binary format is more complex, but the principal function of the loader is the same.

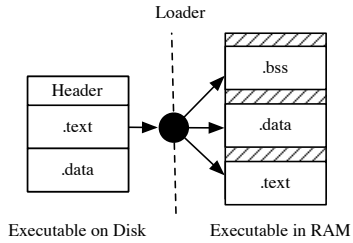

Figure 2.8: Loading an Executable File

There is some confusion in terminology when an executable file is loaded into RAM and prepared for execution. Typically, the term *section* is used to describe a part in an executable file, but as soon as it is loaded into RAM it is called a *segment*. This naming is confusing, especially as the term *segment* is used by some CPU vendors to describe a region of memory which often – but not always – corresponds to segments produced by the loader. Also, in some environments the term *section* is not used, but the different parts in an executable file are referred to as *segments*. Given this terminology one should be careful with the context as the same term can refer to a part of an executable file, a logical memory region produced by the loader or a part of the memory restricted by a CPU configuration.

2.6.1 Static Executables

Self-contained executable files which can be loaded and executed without any external dependencies are called *static executables* or *statically linked binaries*. This type of executable system has a very limited ability to communicate with its environment. For traditional embedded systems this is not an issue as the entire system including the operating system is linked together into a sin-

gle large file which communicates directly with the hardware. For traditional UNIX-like systems each executing binary is separated from others, and its only means of external communication is through the operating system. Any type of communication between different programs (which are not linked together) must be mediated by the operating system using primitives such as sockets, pipes or shared files.

Besides limiting the communication between separately linked files, there is one additional severe limitation with statically linked binaries: there is no effective way to share resources between executing systems. For example, consider a small UNIX-like environment with five different programs running. Each program must have its own copy of all code and data it uses, with the exception of the code and data inside the operating system. This is true even if it is the same system that is executed in five copies (for example one copy per client). This is a huge waste of resources because the vast majority of bytes used by each program are identical.

To allow for more complex binaries with richer communication and more effective sharing of resources,[4] a different type of executable file, called a *dynamic executable*, is used on most mainstream operating systems and even on some embedded targets.

2.6.2 Dynamic Executables

On a conceptual level a dynamic executable is very similar to a static executable. It is a file which can be *loaded* into RAM by a loader and executed by a CPU. In contrast to a static executable, the dynamic is not entirely self-contained but may contain dependencies to other dynamic executables, unknown even to the linker, and which must be combined during the loading process to form an entity in RAM which *can actually be executed*. A well-known example of a dynamic executable file is the 32-bit PE file used in the Windows environment where the main executable file has the extension `exe` and additional modules use the `dll` extension.

A dynamic executable provides three basic features, or abilities, over a traditional static executable:

1. The ability to import symbols at load-time that were unknown when the program was compiled and linked.
2. The ability to export symbols so that other modules can import them.
3. The ability to load modules arbitrarily during execution.

[4] Actually, there are more very good reasons to use dynamic executable formats, but as this topic tends to get somewhat technically complex, we do not dig further into this in this work.

Loading a Dynamic Executable – the Runtime Linker

A loader is a small piece of software placed directly inside the operating system kernel. This asserts that statically linked executables can be loaded directly by the kernel, but it also places a practical restriction on how large and complex the loader should be. To load a dynamic executable, some support is required, typically more than what is placed in a loader. To handle the loading of dynamic executables, a two-stage set-up is typically employed. Firstly, the loader loads and executes a tool called *the runtime linker*, which is a static binary. This tool takes care of the actual loading of the dynamic executable as follows:

1. The basic sections containing code and data for the executable are loaded into `RAM` in a similar way as for a static binary.

2. The loader builds a list of symbols marked for export and where in `RAM` it has placed each such symbol.

3. In addition to these sections the executable contains a list of additional modules it requires in order to execute. Each such module is loaded recursively (i.e., such a module may itself have dependencies to other modules) as described in Steps 1 and 2.

4. Starting with the executable file (the first module loaded), the runtime linker matches the imported symbols (symbols required for that module to execute) with the available exported symbols from other modules. If a symbol cannot be matched, the runtime linking is aborted.

5. Each reference to a symbol is replaced with a reference to the address at which the runtime linker placed that particular object.

6. The runtime linker hands over execution to the newly loaded and linked binary. If additional runtime linking is required at a later stage (for example, if the program loads an additional library), the runtime linker is invoked yet again.

In reality, runtime linking is not as trivial as described in this outline. A runtime linker is actually quite a complex tool. A typical runtime linking problem that is harder than it first seems is when the address range of two modules collides. The runtime linker must support the loading of arbitrary modules at any time, and each module may make essentially arbitrary claims in regard to which addresses it wants to use. To support the loading of two colliding modules, the runtime linker must re-link at least one of the two modules completely – an operation that is both complex and time consuming.

2.7 Executing Software and the Machine

The execution phase is the final form for a software system[5] and where the action is. The executing machinery (a CPU, possibly in conjunction with a virtual machine and peripheral devices) obtains instructions from the software and carries these out – *executes them* – at a tremendous speed.

As we touched upon in the introduction, the only time software can be said to have a *behavior* is during execution. Similarly, the only time executing machinery can be said to have a behavior is when executing some software. Thus, neither software nor executing machinery alone can be said to have a behavior, but this originates from the software combined with the machinery.

To illustrate how the behavior of a system is affected by the combination of software and the executing machinery, consider the following example. A small piece of software is executing on two systems. The systems are identical except the CPU of the first system is 10% faster than that of the second. It is reasonable to expect these different systems to behave differently; for example, one would expect System one to execute faster than System two, although it is not easy to predict by how much. The performance difference of the two systems may cause many other far more subtle differences in behavior from the executing software, such as:

- If the small piece of software includes a real-time operating system, it is likely that the scheduling of tasks will be different between the two systems. This difference may cause certain bugs, such as race conditions, to appear frequently on one system but not at all in the other.
- Should the small piece of software be a video codec, it is likely that it detects that not all CPU power is used and applies more filters to the processing until all cycles are used up. Thus, the final result from the first system differs from that of the second.

When developing a system, one targets an abstract external system (machinery), and this is not identical with the actual executing machinery which eventually will be used. A concrete example of such differences is the Windows operating system executing on commodity PC hardware. The final system may have another CPU, a different graphics subsystem and a divergent set of driver software all contributing to a vastly different executing machinery than what the software was designed for.

2.7.1 The Abstract Machine

An imaginative abstract machine is some idealized version of what can be constructed in real hardware, and its similarity to the actual physical machine

[5] This is with the exception of a crashed system in a post-mortem analysis.

eventually to be used varies largely. As mentioned, when targeting an abstract PC computer system there may be a significant difference between the abstract and concrete machine. For an embedded system, there is less room for differences between the abstract and physical machine.

Abstract Physical Machine

When targeting a physical machine (CPU), the abstract machine consists of some imaginative hardware and adjacent system software, such as an operating system and libraries. The processor architecture is known (such as ARM, PPC or X86) but the exact hardware is typically not. This means that a compiler can generate *correct*, but not necessarily *efficient*, code for the final system.

All modern processors are models of the von Neumann machine, which describes the processors' principal low-level function. The processor sequentially reads and executes instructions from RAM memory, has a set of internal state called registers and has the possibility of communicating with external devices. An instruction can carry out basic arithmetic operations, compare data, and conditionally execute instructions including the *jump* instruction which causes the machine to continue execution at a specified address. There is no principal difference between *instructions* to the processor and other *data* stored in RAM.

When diagnosing a software system, especially for the PC architecture, or when investigating performance issues, the analyst has to consider aspects from the physical machine as the level of detail in the abstract is insufficient.

Abstract Virtual Machine

When targeting a virtual machine such as that for Java or .net, the machine is specified at a higher level of abstraction than a physical machine and allows for instructions that could not be implemented in actual hardware. The principal function of an abstract virtual machine is quite similar to that of an abstract physical one; the machine sequentially reads and executes instructions including comparisons and conditional jumps.

At some point the virtual machine has to bridge the virtual instructions to instructions understandable for a physical, concrete, system and this can be done in a large variety of ways[6]. When analyzing complex issues in a system for a virtual machine, both the abstract and concrete virtual machine and the abstract and concrete physical machine have to be considered. So while the virtual machine creates an idealized machine for developers, the additional level of abstraction is often unhelpful to the analyst.

[6]Sometimes vendors of virtual machines advertise the performance of the transformed code, which is a good example of this.

2.7.2 The Concrete Physical Machine

Irrespective of the number of abstraction layers, such as virtual or abstract machines, at some point there is a real, physical processor (CPU) that executes software. This processor has a number of properties, some included and some excluded in the model of the abstract machine but which all effect the software and behavior of the combined system.

Primitive Types

A CPU has a small number of data types on which it can operate. The most common type is the *native integer*, known as `int` in the C programming language. Other commonly used types are *short integers* and *bytes*. The size of some native types differs between different processors, something that is exposed in the C programming language but typically abstracted in some programming languages. The principal reason for not abstracting the size of primitive types in the C programming language is *performance*. Thus, when programming in the C programming language you get the size which the underlying machine can operate most efficiently on, but you cannot directly choose its size, while in many other languages you get an exact size without regarding the performance during execution. In newer versions of the C programming language, exact sized types are also available (i.e., `uint32_t` for an unsigned 32 bit integer) and can be used if the exact size of the data type is more important than the execution performance. Table 2.2 illustrates some commonly used type sizes for different machines.

CPU / Type	int	short	long	void*
MC68K	16	16	32	32
X86	32	16	32	32
ARMv6	32	n/a	32	32
ARMv7	32	16	32	32
AMD64	32	16	64	64
AMD64	64	16	64	64

Table 2.2: Size of primitive types for different architectures and configurations

There are a few points regarding types and sizes that deserve mentioning:

- *Sizes originate from the* CPU. The concept of differently sized types is sometimes seen as an issue with the C programming language, but the actual size of primitive types is a property of the underlying machine. If you use an environment where the sizes of underlying types are abstracted, you are still vulnerable to problems that originate from the physical machine, as it can never be abstracted.

- `sizeof(int) != sizeof(void*)`. A problem which does correlate strongly with the use of the `C` programming language is the assumption that an integer and a pointer have the same size. This is true for X86 and ARM, but any program developed with this assumption will have severe bugs if executed on a Motorola 68K or on some configurations of an AMD64 processor.
- *Some* `CPU`s *support different configurations*. Many 64-bit `CPU`s such as the `AMD64` can be configured for different primitive sizes. In this case it is up to the compiler to choose which size should be used. While this size can in principle be changed for a single application, doing so is not advisable because it will change the calling convention (see Section 2.4.3), and it will not be possible for such a program to use any supporting libraries and perhaps not even the operating system.
- *Some* `CPU`s *do not support all sizes*. Some `CPU`s such as the `ARM` in revision six or lower do not support operations on 16-bit entities. When compiling for such an architecture, the compiler must find some way around this problem. Typical ways would be to either specify `short` as 8 or 32-bits wide or to generate some form of machine code that operated on 32-bit entities but stored them back as 16-bits. This has a number of implications, especially when working with wide character strings which often are specified to be arrays of 16-bit entities, and this concerns all software systems irrespective of the programming language used.

Alignment and Padding

Data alignment means that a particular data type cannot be accessed at an arbitrary memory address irrespective of memory protection. Most common processors with the exception of the X86 family require some form of data alignment. The most common alignment requirement is called *natural alignment* and means that a type of size X must be stored on an address which is divisible with X. For example, on a system with natural alignment and with 32-bit integers (i.e., `sizeof(int)==4`) an `int` may be stored on address 0 (thus occupying bytes 0..3), 4 (occupying bytes 4..7), etc., but not stored on addresses 1, 3 or 7. It is not possible for a `CPU` to require an alignment larger than the natural alignment, as this would make it impossible to store arrays in `RAM` and hence there can *never* be an alignment requirement for a `char`.

Also for `CPU`s that do not require natural alignment, there is typically a huge performance penalty if addressing data that is not stored on aligned addresses. For this reason most compilers try hard either because of necessity or performance to place data on properly aligned addresses. This becomes evident for the programmer when he or she is using structures in the `C` programming language, as illustrated in Figure 2.9.

In this example, assuming the size of `short` is 2 (16 bits) and the size of `int` is 4 (32 bits), a compiler is likely to insert two unused bytes in the structure

```
1  struct Mine {
2    short a;
3    int b;
4    short c;
5    int d;
6    short e;
7    int f;
8  };
9
10 struct yours {
11   int b;
12   int d;
13   int f;
14   short a;
15   short c;
16   short e;
17 };
```

Figure 2.9: Mine is bigger than yours.

`Mine` after `a`, `c`, and `e`, respectively. Such bytes are called *padding* and are used to achieve *alignment* inside the structure. Some compilers support options to modify the padding used, however if *the executing machine requires natural alignment such requirements are either ignored by the compiler or cause the compiler to generate code that inevitably will crash the system when executed*.

An indirect consequence of *alignment*, *padding* and the use of structures is that there is *no safe way to binary copy data into structures* from a file or other persistent data. The layout of a structure may change if the program is re-compiled, and when compiling for an architecture that uses natural alignment there are restrictions on how the data elements may be placed in a structure.

Performance and Abstraction

The properties of the underlying machine affect every piece of executing software on that machine. If your environment abstracts types, padding and alignment, you will not notice problems as crashes in the executing software but as performance problems. Always investigate the properties of the underlying machine when hunting down performance problems and crashes in software ported to a new machine.

2.7.3 Concrete Virtual Machines and Interpreters

An interpreter or a virtual machine is a piece of software which reads some form of instructions and executes these, thus acting as a machine. Typically the term *interpreter* is used if the machine operates directly on source code

while the term *virtual machine* is used if the machine operates on *object code* or a linked *executable*.

Both interpreters and virtual machines are commonly used in everyday software, and the logic expressed in the interpreted or virtual language typically plays an important role for the larger system. More often than not, interpreters operate on smaller pieces of code (scripts) and virtual machines on much larger chunks, sometimes even a complete system.[7]

To analyze a part of the system which is not compiled to native code, it is essential to determine how that code interacts with the real physical machine.

Interpreters

A piece of code (script) that is interpreted has neither been compiled nor linked. In the worst, and most common, case the loading of such a script simply means that it has been put in a `RAM` buffer from which the interpreter reads its instructions. This implies that two important sources of information and two principal points for intervening with the system are absent (object file and executable file plus linker and loader, respectively). Also, the system is in a form primarily designed for a human developer and not for execution by (any) machine.

To diagnose an interpreted system, one cannot take advantage of the normal tool chain. For smaller systems the brutal analysis approach is often done by trial-and-error (a.k.a trial and terror); a small part of the system is changed until the bug is no longer observable. For larger systems, or to achieve a better analysis, one can attack the actual interpreter and integrate a customized debugger. Examples of the former approach are too numerous to mention, while examples of the latter include ECMAScript (JavaScript) debuggers available in many web browsers.

Virtual Machines

Code interpreted by a virtual machine has been compiled and subjected to some form of linking process. Although the code is not in a format native for a real physical `CPU`, it has been transformed into a format suitable for a virtual machine. From an analysis perspective, this is a huge advantage as both information and the possibility to intervene with the transformation is available to the analyst.

For smaller virtual machines, there is typically no or a very limited debugging support. To debug a system in such a machine, the machine must be attacked. This is similar to an interpreter with the major difference being that the tool

[7] There are a number of notable exceptions to this rule of thumb, such as postscript which is interpreted but usually comes in very large chunks that make up an entire system.

chain is present and provides information and points to intervene with the system. For larger virtual machines there is typically debugging support in the machine which is possible to integrate with a customized or generic debugger.

2.7.4 Recompilation

An alternative to using a virtual machine for a larger system of non-native code is to *recompile* or *translate* the code for the native processor. This has a number of advantages for the system, most notably a significantly higher performance over a virtual machine.

There are two principal approaches to recompilation which have a significant impact when diagnosing a system:

- *Static recompilation.* When recompiling a system statically the output from one compiler or linker is used as an input to a secondary compiler or linker. The final executable file consists solely of native code but no re-compiler.
- *Dynamic recompilation (JIT).* A dynamic or just-in-time recompilation is a technique where a small piece of non-native code is recompiled to native just prior to being executed. The final executable file consists of non-native code and a re-compiler.

Static Recompilation

A system which contains statically recompiled components can *in principle* be analyzed as if the recompilation had not occurred. It is possible to trace the system for performance and use existing debuggers and similar tools as the system consists solely of native machine code instructions.

From a practical view there are some issues which should be considered:

- *Interfaces* . Many programming languages have rather complex interfaces compared to the `C` programming language. Such interfaces, most notably calling conventions, are likely to require much work to decode but contain vast amounts of information.
- *Debug information*. When recompiling through another high-level language, for example when using `yacc`, which outputs `C` source code, debugging information from the second high-level language (in the case of `yacc` that is `C`) is typically made available to a debugger. When recompiling a component at a lower level, debugging information is often lost. The debugging information is, however, available in the first tool chain and should in principle be portable to the second.

- *Large runtime.* Ultra-high-level languages such as `Java` often come with a large runtime library. Information from such a library provides a useful hook into the system and the various interfaces it uses.

Dynamic Recompilation

Dynamic recompilation is a tricky technique from an analysis perspective. The principal problem is that a small piece of code, typically a single function, is recompiled just before it should execute. The resulting piece of native code is kept for awhile should the function be invoked again, but in the case of low memory or bad luck the native version may be lost. Should the same code be invoked again, it can be recompiled to *another* machine code representation and placed at a different address. This causes significant problems for placing breakpoints or even probes, as there is no fixed address for a given piece of code and the only viable code to patch is the non-native code.

When analyzing high-level aspects of a system that uses dynamic recompilation, first investigate if the bug is reproducible without dynamic recompilation. If not, one of the few viable options is to use a low-level debugger.

2.8 Operating System and the Process

Today it is quite uncommon for a larger system to execute as a single task directly on the machine. Most systems execute together with an operating system (OS), and the system is divided into several executing entities, each called a *process*, *thread* or *task*. The operating system provides a number of services to the system, such as the aforementioned ability *to execute in different processes*, *resource allocation* and *access to a file system*.

A modern operating system provides a type of environment known as *multiprogramming* or *multi-tasked* to each process. In such an environment, each process executes quite independently of other processes and there is an illusion that each process has access to the entire `CPU`, memory and other resources.

There are many good books written on how an operating system creates the multiprogramming environment. From a software analysis perspective, the most important aspect of an operating system is understanding the *interfaces* present and the *illusions* created by the operating system and particularly the limitations of these interfaces and illusions. Thus, in this work we will not focus on operating system internals but on the *consequences of using an operating system when diagnosing a software system.*

Operating System Interfaces

Some services provided by the operating system to the executing process are provided explicitly, i.e., not as a part of the multiprogramming illusion. Such

services include *input/output* handling and *resource management*. To use such a service, a process must invoke the operating system explicitly and request the service.

A process cannot request a service from the operating system using a normal function call. This is so because it must transfer control to the operating system in such a way that the protection domain also is changed. Even for embedded systems that have the same protection for the operating system and the processes, a special invocation mechanism is used. This mechanism is called a *system call* or *syscall*. The exact construction of a system call is dependent on the underlying `CPU` but typically consists of a single special-purpose instruction. On some `CPU`s such as the X86 there are several instructions which can be used as a system call, and it is up the the designer of the operating system to decide which should be used. A system call can never be directly invoked from high-level code, as the compiler has no information about which instruction to use or how to transfer parameters. Typically, a small wrapper function (trampoline) is available for each system call. Such a wrapper function is implemented in assembly but is directly callable from high-level code and creates a gateway for high-level programs to communicate with the operating system kernel.

A system call represents one of very few *formal interfaces* found in executable software. There is always a specification on how to communicate with the kernel, and because the kernel has privileges not available to normal programs *there is no way for a program to circumvent invoking the kernel.* This is in contrast to a standard library provided with most development environments. Such a library most likely provides a function to obtain the length of a string (i.e., `strlen`), but a program is free to define its own function for this purpose or to not use zero-terminated strings at all. If a program wishes to open a file, communicate over a network or perform any other privileged service provided by the operating system, it must invoke the operating system kernel, and this can be done in one and one way only: by invoking a particular system call.

This property makes system calls a very suitable target for supervising a program, either to find bugs or analyze some other property of the program. For many operating systems there exist programs (`ktrace`, `systrace`, `truss`, etc.) which log the use of system calls, including parameters and return values, which can later be decoded and used to analyze the execution.

File System

Most operating systems provide programs with an interface to the underlying file system. This interface is provided in the same way as other interfaces – by means of a small set of system calls – but has some special properties that deserve mentioning.

When developing programs in `C` there is typically only a thin calling gateway between the program and the operating system kernel. This means that the

implementation of the underlying operating system is transparent to the program, and special handling may be required in a program that supports two or more operating systems. When developing a program in interpreted or some abstraction-rich programming languages, such differences are hidden from the developer, *except for the file system*.

The file system interface contains much more logic than what is immediately evident to a casual observer. While the calling mechanism to the file system will typically be transparent to programs developed in ultra-high-level languages, this logic is not something which may cause much confusion and be the root of many bugs. Some examples of file system logic (i.e., the semantics of the file system) which have caused bugs are illustrated below.

- *Case sensitivity.* In some environments the file name "foo.out" is semantically equivalent to "Foo.out", while on other systems the two names represent separate files.
- *Deletion of open files.* On some systems it is not permitted to delete a file that is open. On other systems the programs that have the file open still have the original data from that file available, but all new requests to open the file will yield an error.
- *Special characters*. Almost all systems have some characters that are not permitted in file names. A special case is the directory separator, which is commonly slash (/), backslash (\) and sometimes colon (:).
- *Multiple streams.* In some environments a single file may consist of multiple independent data streams, and special magic characters in the path to a file specify which stream should be used. An example in Windows is when NTFS is used as the underlying file system where colon (:) is used to denote the stream to open. In this environment "foo.txt" and "foo.txt:bar" represent different streams in the same file, but "foo.txt" and "foo.txt:" represent the same stream despite the paths being different).

When diagnosing strange behavior in software designed for one environment and ported to another, pay close attention to the operating system interfaces in general and the file system interface in particular. Often there are good tools available for this type of analysis, but this is a well-known problematic area for many systems.

Resource Protection

Most operating systems offer protection of some resources, most notably of memory. This means that a process is not permitted to either read and/or write to/from a particular piece of memory. Should a process attempt a forbidden operation the process will crash.

Resource protection in general and memory protection in particular are used to detect incorrect behavior at an early stage. Many operating systems provide some mechanism for a process to protect regions of its memory against certain operations. In `POSIX` environments this can be achieved using the `mprotect` system call. Memory protection is used by many debugging tools to find problems such as buffer overflows and dangling pointers. While a hardware debugger typically can watch a handful of addresses with a high granularity, memory protection through the operating system typically supports arbitrary amounts of memory but with a lower granularity.

References

[Levine00] Levine, J., *Linkers and Loaders*, Morgan Kaufmann 2000

[McKusick04] McKusick M, Neville-Neil G., *Design and Implementation of the FreeBSD Operating System*, Addison-Wesley Professional 2004

[Solaris05] Anonymous, *Solaris Dynamic Tracing Guide*, Iuniverse 2005

[ELF] Intel Tool Interface Standard (TIS), *Executable and Linking Format (ELF) Specification*, v2 available from `http://www.intel.com`

[PE] Microsoft, *Visual Studio, Microsoft Portable Executable and Common Object File Format Specification*, rev.8 available from `http://msdn.microsoft.com`

[SYSV] The SCO Group, *System V Application Binary Interface*, v4 available from `http://www.caldera.com`

[MACHO] Apple Inc., *Mac OS X ABI Mach-O File Format Reference*, v 2006-10-03 available from `http://developer.apple.com`

3 Principal Debugging

In this chapter we finally step into the core of this work: principal approaches for debugging software intensive systems. We use the term *principal* to emphasize that we will not consider some specific piece of software or machine, but consider only general approaches towards avoiding or controlling *unwanted behavior*.

Before we step into principal debugging approaches, let's reiterate what actually constitutes *debugging*. Previously, we have mainly used the term *analysis* rather than debugging. When a system behaves in an *unwanted* way, we say that this is because of a bug. To fix this issue and prevent the system from behaving in that way, we must firstly identify what is causing the unwanted behavior and secondly modify some part of the program, environment or building toolchain to prevent the system from presenting this behavior.

The first part of this task is the *analysis* of the software system and the second is the *fixing* of the system. The analysis part is essentially always the most difficult and time-consuming of the two.

Layout of this Chapter

The layout of this chapter is as follows:

We begin by explaining why analysis is necessary, i.e. what benefits that can be expected when you perform analysis. *That's the why*.

After that, we will take a look at major tasks for obtaining an understanding of some properties of a given system. *That's the how*.

Then we look at various overarching system states where we can apply our analysis, as well as pros, cons and considerations for each. *That's the when*.

3.1 Why Analyze a System

In the introductory chapter we learned that there are complications to software that we did not – and could not – see during construction, and that some quite interesting things happen when we go from constructing a system to having it execute. As software age both in terms of elapsing execution time, but also in terms of time passed since construction in relation to changes to components it depends on, initial assumptions break and unpleasantries follow.

Those that previously possessed the knowledge of decisions made as to how the system was thought up (designed) and written (developed), may no longer be available: or their knowledge, in any form, may be inaccurate. Other systems may have been constructed around it and can now be considered dependent on its particular behavior. We can say that it has been tightly integrated, or coupled, to its environment.

Now the situation arises that either some previously dormant problem, for any reason, becomes known and needs to be dealt with, or the-powers-that-be require some new functionality to be added. The problem then becomes: what to change and where? Suddenly we need to rediscover secrets that have since long been forgotten or that are just hiding in plain sight, secrets that would allow us to localize a problem, figure out what caused it, devise a fix to the cause, latch on and deploy that new shiny feature or perhaps just ascertain a particular behavior in order to satisfy some external party.

To make these kinds of alterations, we need to understand the system at hand, not only to discover where to change something but also to determine potentially adverse consequences to such alteration.

3.2 Software System Analysis

From a high-level view, the prerequisites for analysis are quite simple: there must be some system and it must behave in an unwanted way. For analysis to be meaningful, it should not be trivial to understand why the system behaves as it does. In principle, it is sufficient that an unwanted behavior has appeared only once to justify analysis, but more often than not unwanted software behavior tends to be *recurrent* and hopefully *reproducible* in a controlled, *repeatable* fashion.

The real task that lies ahead of the analyst is to *explain why a system behaves in a certain way* and give this explanation at an appropriate, or operationalizable, *level of abstraction* such that it – at least in theory – is useful for fixing the system. For example, the explanation *"our system becomes unstable during high load situations"* is inadequate, but an acceptable explanation may be *"our system uses an algorithm which has a time complexity such that when handling hundreds of*

simultaneous clients the computation takes up to ten seconds which causes the watchdog[1] *to restart the system"*. Irrespective of whether the system will be fixed or not, the second explanation is expressed on a level where it would be possible to attack the problem, for example by switching algorithms, increasing the watchdog timeout or simply by using a more powerful machine. The results from a successful analysis do not have to point out a particular solution but merely describe the causal chain leading up to the unwanted behavior in such a way that it can be fixed.

For an analyst to give an explanation at an operationalizable level, he or she must understand the causal chain that causes the behavior. This chain can be rather complex and involves many components and feedback loops unknown when the analysis is started. In addition, because a structured analysis is being undertaken, we tacitly assume that the causal chain is in fact connected to the system[2] and that it is not trivial.

To analyze a system under these conditions, there must be some principal way for the analyst to *refine his or her knowledge of the particular system* in regard to *the analysis issue at hand* by using *general knowledge of similar systems* and *a principled set of tools and methods*. While we acknowledge that there is no such thing as a direct method or principle that outlines a straight path from problem to solution, we argue that one can construct an overlying or abstract method based on principal software properties.

3.2.1 High-Level Analysis Approach

The method we describe herein is an iterative method[3] that we argue is applicable for analyzing a large class of software systems. In this method we extensively use the terms *system view* and *analysis action* which deserve a brief introduction:

The analyst works with a large set of information in regard to the system. This includes results from earlier measurements, expectations on how the system will behave etc. We use the term *system view* to describe the set of information (measurement results, expectations, etc.) an analyst considers at a particular time. During the analysis one may undertake a number of activities to explore the system and its behavior. We refer to such activities as *analysis actions* or shorter *actions*.

With these two terms briefly introduced, let's jump straight into the analysis procedure:

[1] A watchdog is a piece of hardware or software that automatically takes some action if the target system is not sufficiently responsive.

[2] It would not be connected to some adjacent system used by many others where the answer can simply be found using a web search engine.

[3] The process is divided into several steps, and the analyst uses feedback from earlier steps when performing the current step of analysis.

Analysis Procedure

We divide the analysis procedure into the following steps:

0. Start with an initial *system view* based on previous analysis of the system at hand and/or based on generalizable knowledge of software construction and execution.
1. If the *unwanted behavior* can be adequately explained at an *operationalizable* level, the analysis phase is finished. *break;*
2. *Explore the system* by performing some *analysis action*. While certain actions are specific to each and every system, some actions are applicable for a very large class of software systems:
   a) *Subdivide* the system into manageable components (subsystems) so that only a small part, believed to contain the relevant aspects, has to be considered.
   b) *Measure* relevant system aspects (static and dynamic).
   c) *Represent* measured data so it becomes intelligible.
   d) *Intervene* with the system to obtain feedback.
3. Analyze the results from Step #2 to
   a) *validate* representation and system subdivision,
   b) assert that measurements and results from intervening with the system do not have *unacceptable side effects* and
   c) if relevant aspects are discovered, *modify the system view*.
4. Continue at Step #1.

With this overview of method and central terms in place, let's add some detailing to both the philosophical and practical aspects – the *hows* and *whys* of the method. Those less interested in the philosophical aspects can safely ignore the next section, *Analysis Rationale*, and jump straight into the technical *hows* in the following sections.

Analysis Rationale

To understand why we propose the analysis methodology, we will reiterate our view on software and bugs.

We consider the concept of *perfect software*, i.e., software that perfectly follows a completely unambiguous specification, to be an illusion. As discussed already in the introduction to this work, there is no realistic way to construct such

specifications, and neither is there a way for a non-trivial system to adhere to such a specification should one exist.

On the existence of bugs: software has always had bugs, it has bugs today and will continue to have bugs for the foreseeable future irrespective of tools, methods and people.

The analysis methodology we propose has the focus clearly placed on experimentation. We argue that it is through the direct interaction with a system which follows experimentation that an analyst gains knowledge about the system and eventually may be able to explain its behavior. This does not imply that static sources of information are completely discarded, but they suffer from certain limitations that makes them insufficient for extrapolating system behaviors. This will be elaborated on further in Section 3.4.

On reading source code: large software systems are inherently too complex to understand by considering only source code and other static sources.

To explain unwanted behavior, the analyst must understand[4] (relevant parts of) the system behavior. Some analysts have worked several years with the particular system and have a good view of how that system behaves. Given a description of a bug, such an analyst may immediately know where to look for the distal cause. For large systems (with many bugs), relying solely on experience is not suitable. Rather, an analyst must have a principled approach that enables him or her to explain unwanted behavior in a software system without relying on ten years of experience with that particular system. This is important because software systems change rapidly and even an expert may need to rediscover aspects of his particular system.

While each and every software system is unique and has its very own dents and scratches, the way in which we develop software – the programming languages, the tools and the machines – are shared. These shared components form a base, which we model as a series of restrictions on otherwise hard to control execution, and which permits an analyst to apply a series of actions on a software system.

On specific systems and generalizable approaches: we argue that it is possible to generalize debugging approaches and successfully diagnose large systems without relying solely on domain-specific knowledge.

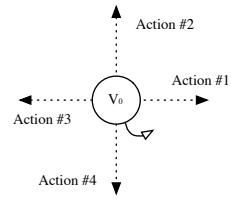

Figure 3.1: Initial View

When starting to analyze a system for a particular unwanted behavior, the analyst has only an *initial view* or V_0 of the system. To advance this view to the next V_1, the analyst applies some *action* on the system as illustrated in Figure 3.1 and analyzes the outcomes. Thus, an analyst who has a vast experience with the system starts with a more informed initial

[4]In other words, he or she must be able to explain at an operationalizable level.

view than an analyst who is new to the system. However, by applying analysis methods that operate on a class of systems, it is possible to start the analysis (i.e., the transition $V_0 \Rightarrow V_1$) in a safe manner without relying on expert knowledge of the particular system.

The analyst works by iteratively applying an action on the system and analyzing the outcomes. As this procedure progresses the analyst successively obtains more information about the system and can deploy specialized and more complex actions. At some point the analyst has narrowed down the analysis to a point in the system where some consequence of the ultimate cause can be observed and the analyst can thus explain the behavior of the system.

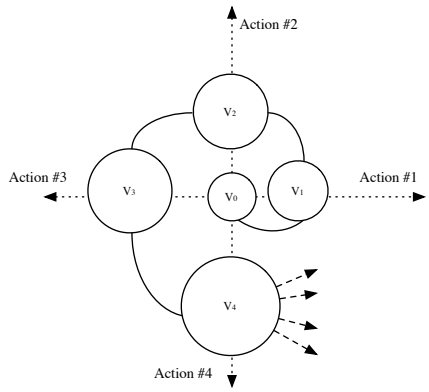

Figure 3.2: Views and analysis actions

When following an experimental pattern of system exploration, there is no way of guaranteeing a straight path from the initial view to the final, i.e., the view where the analyst can explain the behavior. It is possible, and even likely, that some steps will not bring the analysis forward but yield empty results. Actually, there is no way to guarantee that the analyst will be able to explain the behavior at all. He or she may fail because of lack of information, skill or both.

The principal advantage of using an experimental approach is that one works with the actual behavior of the target system. This enables the analyst to establish a feedback chain where he or she measures properties inside the system he or she manipulates. This is a large advantage over, and in stark contrast to, methods that consider only source code and/or other static properties of a system.

While there is no way of guaranteeing a successful analysis for any methodology, we argue for an experimental approach to software analysis in general and for our methodology in particular. We argue for an experimental approach as it captures not only static aspects of software but also the important dynamic aspects. We argue for our methodology as besides being based on an experimental approach it also permits a high level of generalizability because the main actions are tied to properties which originate from programming languages, tools and the machine shared between a large class of systems.

3.3 System Views and Analysis Actions

During the process of analyzing a system, the analyst has access to a vast amount of information about the system and its behavior. The information comes from a variety of sources both static and dynamic, but the majority will be deemed irrelevant or superfluous and promptly discarded[5]. Also, the analyst has some *expectations* on how the system and/or some component will behave in some particular situation, and possibly some opinion on how the system can be subdivided into smaller entities.

We use the term *system view* to describe all aspects of a system the analyst actively considers (and considers relevant) for the particular task at hand. This will include many different aspects of the system and change significantly as the analysis progress. We focus on three such aspects which relate closely to our model of analysis actions and that we consider of particular relevance: *expectations* on the system and its behavior, results from *measurements* and ideas of how measured entities can be given a suitable *representation*.

Expectations

When working with a system, the analyst has a number of expectations on the system and its behavior. Expectations vary largely in terms of vagueness from initial assumptions (”when executing the system with the specified parameters it will not behave as desired”) to concrete falsifiable hypotheses (”a failure to initialize this particular pointer causes a crash as it is dereferenced and has the value `NULL`”).

In some debugging texts every type of *expectation* is described (modeled) as an *hypothesis*. While it may very well be possible to model every expectation or belief one has as a series of hypotheses, in our view this is not a particularly useful[6] way of addressing software analysis. Rather, the pragmatic analyst has to accept initial expectations which cannot easily be quantified and, as the analysis progress, form quantifiable hypotheses about the system and its behavior.

For systems which the analyst is not particularly familiar with, the following list provides some starting point for formulating expectations of the system:

- *There is likely a causal relation between proximal and distal cause.* Some crashes, like dereferencing a `NULL` pointer, creates a crash that clearly reveals the proximal cause.

[5]For example, the analyst is likely to have access to the entire source code for the system, but this is too much information to consider as a whole.

[6]Starting with ”I suspect a problem in the system” (true) followed by ”I believe it is in some component” (true) and ”this component may be identified” (true) is not a practical way to analyze large systems. Rather, the analyst will have to consider complex causal chains in his or her expectations that are not necessarily suitable to divide into a (very large number of) binary quantifiable hypotheses.

In our experience more often than not there is some form of trace, path or discoverable flow from the proximal to the distal cause, and it is a reasonable for an analyst to *expect* the existence of such a link. Put in a somewhat more operationalizable form: it is reasonable for an analyst to expect the existence of clues that reveal the distal cause close to the proximal cause of a crash.

- *The system logs and/or traces reveal usable information.* Many systems have some form of logging facility that may reveal clues about the state which caused the system to behave in an unwanted fashion.

  Perhaps more important than logs is the output from tracing tools that provides automatically generated flow logs, such as the communication between a program and the operating system kernel[7]. Automatically generated logs are typically of a higher quality than manually generated ones as a tool is less likely than a developer to ignore seemingly irrelevant aspects.

- *Components that got the chance to execute are more likely to cause havoc than those that did not.* A good practice for forming an initial expectation of which components (subsystems) are involved in unwanted behavior is to determine which components that actually were executed. While it is possible the non-executing components affected the execution, it is much more likely that components which actually executed have a central role for the bug.

  For certain systems this observation may seem trivial, like when discovering a bug in Program #1 it is quite obvious that the analysis should not begin with investigating Program #2 just because it is stored on the same hard disk and happened to execute at the same time as the first. For large monolithic systems, however, it is not as easy to determine which components actually affect each other and – we argue – it is relevant and non-trivial to determine which components that actually were executed. It a reasonable expectation that those that did are more interesting to investigate than those that did not.

- *Certain functions are bug magnets.* Even if the analyst is not familiar with the particular system, there are some kinds of functions that are more error-prone than others. Examples include parsers, code executing concurrently in multiple threads/processes, low-level input/output routines, and optimized implementations of complex algorithms.

- *Certain groups of developers never do right, always do wrong and never learn.* For systems with a large legacy, it may be a good investment of time to ask a colleague about which developers or groups of developers that have previously failed to meet expectations. If they have failed once, they can fail again.

[7]Please refer to Section 2.8 for a discussion of this interface and why, by necessity, it can always be traced.

Measurements and Representation

Having access to accurate and relevant measurements is a crucial asset for the analyst. This is clearly reflected in the large set of debugging tools which work around one or a small set of measurement points. For example, the function of a debugger, tracer and profiler is based on a particular measurement inside an executing software (i.e., a *dynamic* tool) and there exists a plethora of tools that analyze particular *static* aspects of source code to find alleged unsafe, strange or otherwise disliked constructions.

A human analyst has no sense with which he or she can interpret (sense, feel or hear[8]) a measurement from a software system. To analyze such measurements, these must be presented in some form that makes them intelligible. Finding such a way to present data is often called *representing the data* or *finding a representation* for the data, a task important for the analyst to master.

During analysis a large number of measurements are taken and the measurements considered relevant typically change as the analysis progress. Similarly, the analyst may very well need to find new representations to make earlier measurements intelligible, which is required to draw some conclusion from the collected data.

Together with expectations, measurements and representation are fundamental mental building blocks for how the analyst considers the system at a particular point in time, what we model as the analyst's *system view* for that particular time.

3.3.1 Taking Actions to Discover the System

For the analyst to expand his or her *system view*, he or she takes certain *actions* on the target system. A particular action may be initiated for many different reasons: to test an expectation (perhaps even to falsify a hypothesis), to obtain additional measurements from a subsystem or to observe how the system behaves in some special case.

When analyzing the results from a particular action, the analyst obtains new information about the system and updates his or her system view. Put another way, by applying some analysis action the system view is changed from V_n to V_{n+1} where newly discovered aspects of the system are integrated. After a successful series of actions, the analyst has obtained a system view sufficient to provide an operationalizable explanation of the unwanted behavior. The execution of different analysis actions and the corresponding system views are illustrated in Figure 3.3.

By using this notation we do not imply that there is a straight path for transforming nonexistent domain-specific knowledge into operationalizable explanations of complex system behavior, neither do we imply that every action

[8]Though we hear arguments that truly bad code stinks, we interpret this in a figurative way.

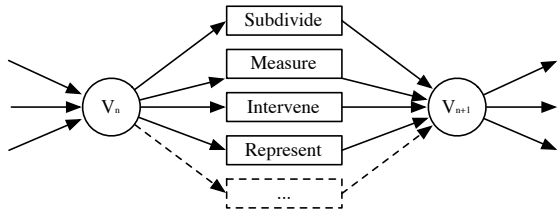

Figure 3.3: Discovering the system by experimentation.

applied to a system results in relevant, usable knowledge. However, what we do claim is that there are certain generic actions which can be applied to a software system – not because of how the particular system is constructed but because of restrictions placed by the compiler, `CPU` and other components – that reveal information specific to the particular system and that can be used to analyze a software system. It is by applying such actions in conjunction with system-specific actions that the analyst discovers the distal cause of an unwanted behavior.

3.3.2 Action #1: Subdividing a System into Components

Central to analyzing a complex system is subdividing the system into analyzable communicating subsystems/components and the communication between such components. This is key because every non-trivial software system is, by definition, too large for an analyst to fully comprehend.

By dividing a system into subsystems, it is possible to focus on one distinct subsystem at a time but, perhaps more importantly, it permits instrumentation at borders between components rather than modeling the entire system as a giant, impenetrable black box.

We can, due to the way software is constructed, always divide a software system into one or more system of subsystems and partitioning a system in ways befitting of the particular problem at hand is one of the primary challenges for an analyst to deal with.

In Figure 3.4 a black box view of a system is illustrated on the left hand side, and the analyst's view of communicating subsystems on the right hand side. Given this illustration it is not obvious if the boxes represent functions, source code files, operating system processes or nodes in a networked system or all of these. It is the job of the analyst to divide the system in a way befitting the problem at hand.

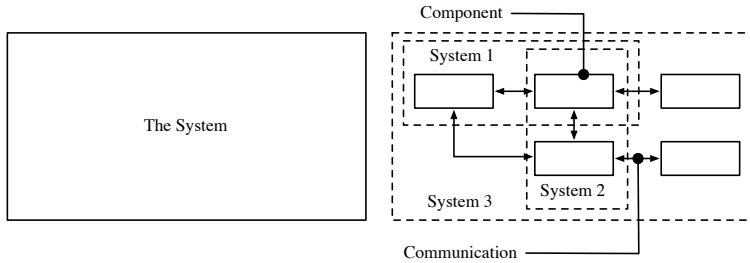

Figure 3.4: "The System" monolith is subdivided into communicating subsystems

Modular Systems

Some systems, such as client-server or peer-to-peer based, are often considered easy to divide into obvious components, and the first step taken when diagnosing such a system seems to be to finding out in which part the bug exists. While subdividing into seemingly obvious components is not necessarily a bad thing, it is important to keep in mind that a system can be divided in several ways and that a bug which manifests itself in one part may exist in others as well.

Monolithic Systems

From a theorist point of view one may argue that a compiled and executing software system is largely a black box and that one has a very limited insight into its internal structure. As we described in the last chapter, this may be true from a view of *what is possible for a* `CPU` *to execute* but has *no relevance for real software systems whatsoever.*

A modern executing system can be modeled as a series of communicating subsystems, each communicating with its environment using some form of protocol. Some of these protocols, or interfaces, are hard and cannot be circumvented, others are chosen by the compiler and linker and are thus predictable and yet others may require additional labor to unfold (see Section 2.4.1). Because of the way modern software is constructed, it can be divided into subsystems which we can – at least on a conceptual level – isolate and conduct experiments on. For practical examples of how linked entities can be divided into subsystems please refer to the previous chapter, particularly the sections regarding calling conventions (2.4.3) and the operating system interface (2.8).

3.3.3 Action #2: Measure

A substantial part of analysis consists of *measuring* different properties of the target system. By measuring, the analyst obtains information about crucial aspects such as the execution state or memory layout which, combined with other properties, provide a useful clue to explaining the unwanted behavior.

Principally, every aspect of a software system can be measured given the right method, tool and sufficient amount of time. This is true because the sources of the system are available (at least in the form of machine code), and the lowest-level machine (the `CPU`) can be probed to reveal the execution path. Trying to measure each and every aspect of even the tiniest system is clearly not a good idea simply because the sheer size of the static and dynamic domains (machine code and execution log). What is important, however, is the principal ability to measure and the two domains or sources of information.

The first source of information is the actual source of the system or program: the source code, object code and binary file which eventually will be executed. We call information from this source *static* as it contains elements of the non-executing, or dead, system. The other source of information originates from the system execution and contains properties such as execution path and values of variables etc. In contrast to static information, this describes aspects of the executing, or live, system and we call data obtained from this source *dynamic*.

Both static and dynamic measurements are important assets for the analyst who should explain complex aspects of a system. However, from a measuring perspective these two sources of information are vastly different and thus we consider them separately in this discussion.

Measuring Static Aspects

Every aspect of a software system that does not strictly come from its execution comes from some source and is what we describe as static data. The most well-known source of static information is *source code* which is transformed by the toolchain to form machine code eventually to be executed by the `CPU`. In addition to the source code, there are other sources of static information, such as *runtime libraries*, automatically inserted by the toolchain, and artifacts from the compiler. Some important static aspects of a software system are:

1. Instructions for how the system should behave when executed
2. Values of constants
3. Data structure layout

When discussing static aspects it is important to stress that these typically *exist in several forms simultaneously*, and the particular form that an analyst is interested in depends on the issue and the progress of the analysis. For example,

"instructions for how the system should behave when executed" can refer to parts of the *machine code*, to the *object code* for a particular function or to the *source code* originally written by the developer. All these entities represents the same abstract "instructions" but are in different forms closer to the human developer or the executing machine.

When an analysis is started, relevant static data is essentially always in a form suitable for the machine rather than the analyst[9]. This is so because if some unwanted behavior has been observed we know that the system is or has been executing (as only executing software can have a behavior). As we recall from the previous chapter, this means that a toolchain has been applied to a number of source code files and a linker has combined the outputs from the compiler into something a loader has put into memory in an executable form.

Because the software is not in a format designed primarily for a human to understand, but for a machine to execute efficiently, the static aspect of interest may have been transformed into a shape very different from how it was originally defined. For example, if the analyst is interested in a function call to `memset` or the layout of a compound data structure (a `struct` or `class`), he or she may find that the compiler has replaced the function call with a suspicious machine code idiom[Gaines65] and that the data structure has been filled with dummy elements for optimization.

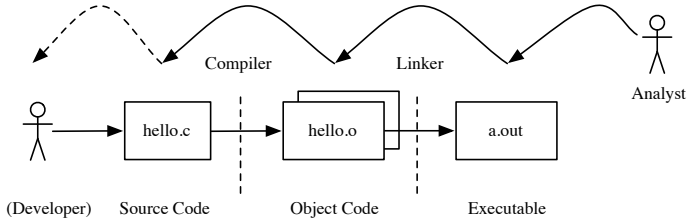

Figure 3.5: Measuring static aspects means reversing the toolchain.

Measuring a static aspect of a software system at an operationalizable level of abstraction (i.e., where it is usable for someone) means that the analyst must follow a reverse path through the transformation toolchain from the executing software where the unwanted behavior occurs to some static source where it makes sense. This source may be the original source code but also any other step in the chain, as illustrated in Figure 3.5.

For an analyst to translate what happens in the execution into an understandable static entity, he must have a good view of both *the machine* and *the trans-*

[9]The exception would be an interpreted system such as a postscript program which has only one form. In the particular case of postscript, we leave it up to the reader to tell if the format is suitable for a human being or a machine, as we're not quite certain and it seems to us as neither.

formation toolchain. Because the transformation toolchain is designed for a one-way operation the other way (from source to machine), this can be quite a daunting task. For details about a typical transformation toolchain please refer to the previous chapter, Software Demystified.

Common Static Aspects of Interest

Some static aspects are often used and deserve a special mention:

- *Instruction to source code line.* A common task is to map a crash at a specific instruction to a line of source code in a particular file. For many systems there are automated tools which assist this step, but the principal approach is as follows:
  1. Identify which *symbol* occupies that particular address.
     This information concerns the memory layout of the system as a whole, and is thus available to the linker, *but not to the compiler*. Typically this information can be found in a debug map over the executable file.
  2. Identify which *object file* contains the particular symbol.
     Typically this information is saved in some debug database for the system, but if no such database is available each object file has to be analyzed for the exported symbol[10].
  3. Identify the offset from the start of the symbol to the particular address.
     The internal structure of the symbol is up to the *compiler* and given an offset it's possible to match the instruction to a source code line from the particular file. The exact matching from symbol+offset to source code is dependent on the compiler, but typically all of the following approaches work:
     - Many compilers provide a means to produce an assembly output where the offset can directly be matched to the instruction and source code line.
     - Most object file formats contain an automatically generated debug section which specifically contains this mapping information (as it is required by a symbolic debugger).
     - If neither assembly file generation nor debug information is available, the analyst can add some easily identifiable code to the function (such as a call to a function pointer in `C`) and move that around in the function to approximate the mapping between assembly and source code.

[10] The symbol has to be exported from the object file as otherwise there would be no way for the linker to find it.

- *Memory to variable.* To translate a memory address into a variable (i.e., find out which variable occupies a particular address) one must first determine if the address originates from a dynamic or static area. Typically, there is only one large dynamic area, which is the memory managed by `malloc` and all other variables comes from some static area.
  - *Static* Identify the symbol and the offset in the same way as for functions. The symbol identifies the start address of the variable and the offset is, unsurprisingly, the offset into the variable.
  - *Dynamic* If the memory originates from a dynamic memory allocator (i.e., `malloc` in `C` or one of the `new` operators in `C++`), it is not possible to determine which variable is stored on that particular address using only static information.
- *Data structure layout* The layout of complex data structures is processed by *the compiler* and is essentially not in an analyzable format after this transformation. Information about data structures (most notably `structs` in `C`) is most efficiently analyzed after preprocessors but before the compilation of a source code file.

Measuring Dynamic Aspects

We describe all aspects of a software system that originate from the *execution* as dynamic. This includes *behavior*, both wanted and unwanted, the value of a variable at a particular time and the actual execution flow inside a part of the system or even the system as a whole. A dynamic property is never directly created in a source code file or by a tool in the transformation toolchain, but comes as a consequence of the fully transformed static properties (i.e., machine code) and various inputs (such as user interaction) with the system.

What makes dynamic properties so important is that every type of software behavior is a dynamic property[11], and thus the actual unwanted behavior that should be explained is a dynamic property. What makes dynamic properties so difficult is that there is no way to directly create them.

There is a large variety of tools that work with dynamic aspects of software systems at different levels of abstraction. Such tools include:

- A symbolic *debugger* – which displays the value of a variable.
- The *kernel call trace* – creates a log of the communication between its target and the operating system kernel.
- A *tracer* – that records the exact execution flow in a system.

[11] This is true because only executing software can have a behavior at all, and all aspects that originate from execution are, by definition, dynamic.

- The *network packet recorder* – saves data from a client-server based system.
- The infamous `printf` or a similar function – appends data to a log.
- The *flashing of a LED* to signal information about various low-level states.

Measuring dynamic aspects is very different from measuring static. While measuring a static aspect is more the work for a detective resting in his or her comfortable armchair trying to make the puzzle fit, dynamic analysis means interacting with an executing target where every interaction may in itself affect the target being analyzed. This interaction with the target is necessary as the actual properties from the execution are of interest and there is no way to measure these without having the system executing.

Side Effects

A tool that inadvertently affects the system is said to have an unwanted *side effect*. A side effect can either be *direct*, meaning that the tool directly affects the system (for example by destroying internal data), or *indirect*, when the tool affects some part of the causal chain that causes the system to behave in a different way with the tool present. Direct side effects are comparably easy to understand and manage; it is essentially a *bug* in the tool which itself can be diagnosed and fixed. Understanding the implications of an *indirect* side effect is significantly harder.

Ignoring Indirect Side Effects

Some dynamic tools have well-known side effects that are inherent in the function of the respective tool. For example, a *debugger* permits an analyst to pause the execution of a thread and will inherently affect the timing of its target. This lies in the function of the tool as such, and there is no way to construct a debugger that can sidestep this effect. This does not, however, imply that a debugger has to impose a performance impact if no breakpoint is hit. Most debuggers have such side effects too, and these are not inherent in the tool but could, at least in theory, be fixed with better debuggers.

While timing is probably the most commonly imposed and ignored side effect, there are others. Tools that collect data for later analysis typically use memory resources that would otherwise be available to the target, which makes it hard to predict the indirect side effects in rare low-memory situations.

To get a better insight into side effects imposed by analysis tools in general and debuggers in particular, look for discussions on software obfuscation. The practitioners in this area are dependent on making software analysis hard and often provide valuable insight into the inner workings of dynamic tools. This, of course, is very beneficial for the analyst.

Minimizing Indirect Side Effects

In some cases it is essential for the analyst to not only be aware of and ignore but to actually minimize indirect side effects. Typically, this is required when *timing* or *resource usage* of other tools is an issue, and the analyst has to construct some specific tool to address a particularly tricky situation.

When constructing dynamic tools a number of properties from the *underlying machine*, *the toolchain* and the concept of *system subdivision* can be exploited by the analyst to minimize the side effects.

A good understanding of the machine is important in order to minimize side effects because as the level of abstraction in the probe[12] increases, the chain of dependencies for the probe increases dramatically. For example consider a simple probe which prints a diagnostic message with the value of a variable at some specific time. If this probe is developed in some high-level language like `C`, it will have a dependency on the entire toolchain, to runtime libraries, and one cannot guarantee how it will behave in special conditions such as ultra-low memory or with interrupts disabled. If the probe is written in assembly or, even better, in raw machine code, there are no such dependencies and the behavior under special conditions can easily be assessed.

When working with ultra-high-level languages that require a virtual machine, a slightly different approach can be taken. Rather than developing the probe in byte code, the analyst can target the entire virtual machine for example by means of developing a custom emulator that permits low-level analysis of the target system. This provides a high degree of execution control with few side effects for the executing system.

Probing the execution inside a program at an arbitrary point is hard. To minimize the indirect side effects, it is preferable to place the probe at a very low level of abstraction, which, for compiled systems, effectively means in machine code. Developing a probe at this level of abstraction and placing it inside the target is not a trivial task.

Because the flow of instructions is likely to be highly optimized, there is a risk that a probe at such a low level of abstraction will create both direct and indirect side effects, which the analyst will have a hard time predicting. Thus, it may seem as if there is a contradiction in the level of abstraction, which is beneficial for the placing of a probe; on one hand we want to place the probe at a low level of abstraction as the probe itself will have fewer side effects, but on the other hand we want to place the probe at a high level of abstraction to avoid having to consider complex optimizations and machine code constructs placed by the compiler and that a probe easily may affect.

As we have already learned, there is no need to consider every form of execution possible on a given `CPU`. This is so because of restrictions placed on the executable file originating from the programming language, compiler, linker and

[12] A probe is a dynamic tool too simple to rightfully be called a tool.

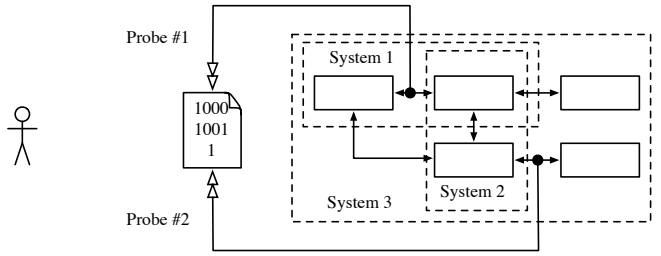

Figure 3.6: Using Probes #1 and #2 the Analyst Measures Data From the System(s)

other components in the transformation toolchain. This knowledge is highly applicable in the domain of dynamic measuring.

We argue that by (epistemically[13]) dividing a system into communicating subsystems with well-understood borders and protocols, the communication between such subsystems can safely be measured by an analyst with minimal risk of causing unpredictable, unwanted, side effects.

Common Dynamic Aspects of Interest

Perhaps with the exception of `printf`, the traditional debugger is the most commonly used and well-understood dynamic tool, and the aspects of a system visible through the debugger are sufficient for many types of analysis.

In some cases more complex dynamic measurements are required, such as when side effects are an issue or when the target is executing in field and the debugger cannot follow. In these cases simple logs of execution such as values of some key variables during the execution are commonly used.

As a side note to this quite mediocre use of dynamic tools, the authors wish to stress that dynamic tools and dynamic trickery can be used for quite interesting measurements and offer a great flexibility many analysts miss or ignore.

3.3.4 Action #3: Represent

There is no way for a human analyst to directly understand measurements from a software system. No human analyst has a sense for bad tree balancing,

[13]The system does not *actually* have to be constructed in this way. It is sufficient if the analyst can construct a model in which it can be viewed in this way.

incorrect parameters or the use of `NULL` pointers. To understand measurements from a software system, the analyst must find some way to read, visualize or project the data such that it makes sense. The process of making data understandable or intelligible is often called finding a *representation* for the data and is an important task not only for the software analyst but for anyone working with science.

Representing Primitive Types

The simplest form of representation concerns the internal format of primitive data types for a particular machine, such as how integers, strings and floating precision numbers are stored in memory. Understanding this representation is important, but for an analyst to make sense of complex measurements a representation not only of primitive types but of complex types and execution flow is required.

Generally, when discussing representation we assume the analyst to be familiar with the representation of primitive types and we refer to the construction of representation models for complex data structure and execution flow.

Reversing the Toolchain

A commonly used way of representing data is to use the original descriptions from before the toolchain was applied. This means that the entire toolchain is reversed, and the analyst is presented with a view identical with, or similar to, the developer's view from when the source code was originally written. Thus, rather than being presented with machine code and binary encoded structures, which is what the system looked like when it crashed, the analyst is presented with a view that consists of high-level source code and friendly names for data.

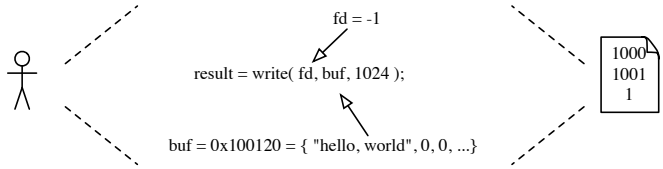

Figure 3.7: The System State Becomes Intelligible When Using a Proper Representation

This type of representation is used by several types of tools such as *high-level debuggers*, some views in a *tracer* and a *decompiler*[14]. Because of its wide use,

[14] Actually, creating such a view is the only task for a decompiler.

the representation is well-known to both analysts and developers and provides a view where many complex properties of the toolchain are abstracted. Using this representation provides *rapid debugging* (compare with rapid development) that permits analysts or even developers to swiftly identify some kinds of behavior within a short timeframe.

The large disadvantage of this form of representation is essentially the same as the large advantage: it abstracts large quantities of information and this may shadow important aspects needed by the analyst. In our opinion this form of representation is particularly ill-suited for situations where:

1. The distal cause is strongly related to some tool in the transformation toolchain or to the machine. As an example consider a software which does not properly handle structure alignment and padding. Such a program will either crash the machine or get incorrect data during execution, but there is no way to *represent* this problem in a source code view.
2. The data is not easily observable in the source code at all. While source code is a powerful way of representing program flow, it is not necessarily a powerful way to represent the data processed by the source code.

When doing complex analysis, especially when working with data rather than flow, a non-native way to represent data can prove a useful tool for the analyst.

Non-Native Representations

The other main approach to representation is using some format designed specifically to aid the work of the analyst. This means that the data is not necessarily represented in some form it once had, but in a form arbitrarily chosen with the sole purpose of aiding analysis.

This approach is commonly used for measurements from complex data structures where the layout of the data structure significantly affects some property but the source code does not provide a good view of this property. An example of such a situation is a binary tree, illustrated in Figure 3.8. The depth of the tree determines the average time required to find an item, but the depth of a tree is not as easy to observe in source code as in a graph.

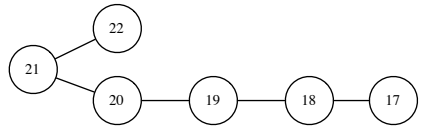

Figure 3.8: No Bonsai.

In Figure 3.8 it is quite obvious that the tree is not balanced and thus an analyst can easily tell that search operations on this tree will require more time than operations on a balanced tree would.

For data structures there are often generic representations which the analyst can borrow for his or her own analysis. For very complex analysis of a system, especially if custom-developed dynamic probes or complex data structures are analyzed, specialized representations for measured data may be required. Some ways of representing data may also require support from additional tools to generate the final entity which is usable for the analyst.

3.3.5 Action #4: Intervene

Intervening with a system is the real core business of experimentation and provides strong feedback to the analyst. When intervening, the actual behavior of the system *with and without some modification* can be studied and compared. Because of the comparably high reproducibility of unwanted behavior in software and the possibility of subdividing a system into smaller parts, intervening with a system can be used at many different levels of abstraction and provide the feedback required for the analysis to progress. Concrete examples of intervening with a system include:

- Using a debugger to step over parts of a presumably incorrect function *provides feedback as whether that part really was incorrect.*
- Injecting faults into an executing system *provides feedback on system stability in hard-to-reproduce situations.*
- Statically replacing parts of a system with dummy implementations *provides feedback on which part of a system that contains a particular bug.*
- Replacing one implementation with another *provides feedback on if and how that particular function affects some behavior.*

When intervening with a system there is a clear risk that the analyst causes more harm than good. The analyst has not yet a good view of the system's function (or there would be no need for experimentation), and intervening inevitably means meddling around with the target system's internal structure. Intervening with a system, like measuring, can be done using two different approaches: *dynamically* inside the executing software, or *statically*, in some source.

Intervening with the static properties of the system means that the analyst modifies one or more software source (typically source code), rebuilds and re-executes the system. This approach combined with some primitive logging is a commonly used, and rather ineffective, way of diagnosing simple systems.

It is possible to achieve more complex analysis using a static intervention approach, for example by replacing functions, modifying a memory map or switching parts of the toolchain. Intervening with static aspects of the system is quite similar to traditional software development, and thus nothing we will further elaborate on but tacitly assume the analyst to be familiar with and consider trivial.

Intervening with Dynamic Aspects

Intervening with the dynamic aspects of a system means directly affecting the execution. This approach is used in debuggers where the analyst has the ability to modify the value of a variable or change the flow of execution. Intervening directly with the execution is a powerful action to take, but also one that comes with the risk of significant side effects.

When using a debugger the analyst views the system through a software microscope. He or she observes values of individual `CPU` registers and sees the software as a stream of machine code instructions which execute one after the other. In a symbolic debugger, rather than registers and machine code he or she observes variables and high-level code but still on a microscopic level. Modifying the value of a register or local variable is comparably safe; the effects are local and can easily be assessed by the analyst. Dynamically intervening with systemic properties, for example by doing fault injection into an executing system, hijacking functions or transparently patching return values, has a larger impact and the side effects cannot easily be assessed.

To intervene with dynamic properties that are not on a microscopic level, the analyst must find a way to ensure that the action does not cause unacceptable side effects for the system. With the system only partially known this is not an easy task, but the same principles used to identify borders when measuring inside a system can be used to intervene with the system.

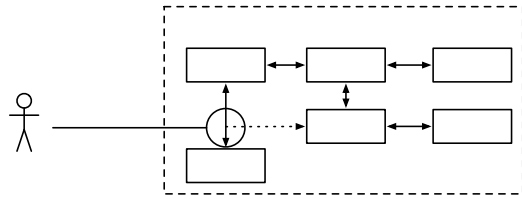

Figure 3.9: The Analyst Intervenes with the System Behavior

We argue that the same principal borders that are used in a system when measuring dynamic data can be used to intervene with the executing system. Similarly, to intervene with the system through the toolchain (i.e., with static properties of the system)

entities which originate from the building toolchain, such as language artifacts, can be used to allow safe intervention.

3.3.6 Other Analysis Actions

We have described a methodology with four concrete actions an analyst can take to analyze a specific system. These actions, and our methodology as such, is clearly in an experimental framework. It is experimental as we claim that it is by *actual experimentation with the system that the analyst gains knowledge about the system* – knowledge which is refined as the experimentation progresses and that eventually is sufficient for explaining the unwanted behavior of the system.

In such a framework there is always room for additional actions and feedback loops, should these be beneficial for the analysis of the particular system. We would strongly encourage an analyst who uses this view of a system to continue subdividing, measuring, representing and intervening as well as taking other actions that seem to fit the particular analysis task at hand.

3.4 Information Sources

Our interventions are fueled by the understanding we have of some system. When this understanding is insufficient we need to refine, and perhaps re-evaluate, the whole or parts of this view. There is also the need to review and verify that the consequences from an intervention are the ones desired; all in all we want to improve some situation and definitely not worsen it. To perform both of these tasks, we need more information about some specifics of the particular system. Mentioned earlier, we look at state in various forms with the most detailed concerning the executing machinery and all of its stores of data involved during execution (machine registers, primary, secondary, tertiary storage and so on) which is immensely far beyond the point of human comprehension. Because state is highly involved in the presence and occurrence of a bug, we need to somehow reduce this set so that it contain only (or mostly) the relevant ones for a particular case and represent them in a comprehensible way. If we consider the entire chain of managing a system and try to categorize the state space into large, distinct groups, we can see that a few such subdivisions are different in what we can do and what we can learn. The categories selected for discussion here are *Pre-Execution*, *In-Execution* and *Post-Execution*.

3.4.1 Pre-Execution

Review again the chain through which software gets pieced together. Code gets written by someone or something. This code is run through an (optional) build

system, producing rules for regulating how the rest of the chain is to behave and making sure various tools are used in a particular order. A compiler transforms the code into a form manageable by the target machine, and a linker puts the resulting pieces together to form some kind of tightly knit digital blob symbolizing the end of the pre-execution state. At this point, there is already an abundance of potential sources for information, and as previously mentioned, these are the ones most readily automated. Such automation we call static analysis.

It is tempting to accept the notion that all information available at this stage is both necessary and sufficient to determine and adjust everything that could ever be known about – or happen to – a system. But already during the dawn of the theoretical foundations to computing, the ultimately undecidable problem of determining if a program will ever finish was described, a problem which also correlates to determining other states using program code alone, and the view of a computer program when these ideas were coined was far more simplistic than the software intensive systems that we are now forced to deal with. There may well be components of an entirely different flavor as parts of the mix.

Let us quickly run through the main information sources along with their respective use[15]:

Design Documents

It would appear that there are more types of design documents than there are actual designers. Roughly speaking, all these different types can be summarized as either wordy or boxy, employing various forms of notational and/or rhetorical trickery to hide the fact that they purposely lack the precision needed in order to be usable by themselves. To assist this cause, there are methods for validating, simulating, profiling and proving their correctness, all of them with a very limited relevance for the actual system at hand.

Design documents may serve as a starting point for an investigation – in order for the analyst to get a broad overview of how someone at some time intended the system to hold together – but for systems with a large legacy there is typically a significant discrepancy between what's stated in the design documents and how the system actually is constructed.

Source Code

Source code is the first formal description[16] that has enough precision to either directly – or through some transformation – form each individual component

[15] This is from a debugging point of view.

[16] Smaller components from mathematical models of smaller components, algorithm proofs and so on, are not directly convertible into executing software.

of the intended system. A large portion of software problems can therefore be traced back to this point and this is a desired overarching task when debugging. The alternative, i.e., situations where for some reason the source code is irrelevant or missing, suggests either a more complex or a mechanical problem (e.g., failing hardware). Some of the initial problems of using source code as the starting point for investigation is a) that it might not be yours, meaning that whoever wrote the code managed to miss something and chances are that you will too and then start questioning even the things that are working correctly but written in a way that deviates from your style, and b) that it quickly expands in scope and size so that the set of statements to consider far exceeds the capacity of rational thought. With this said, source code will still be the primary target for making corrections; targeting something more dynamic, while not unheard of, should be an exception rather than the norm. Therefore, we need some way of mapping a connection between some runtime observation and source code. The more invasive way would be a separate build with debug outputs, and the less invasive way would be tracking the mapping between symbol names and memory addresses.

Build System

Build systems have a nasty habit of growing more complicated than even the source code it is supposed to work on, and quickly becoming an integrated software component in and of itself. Determining interdependencies between source code modules, passing the right parameters onto compilers and other static tools maintaining support for various configuration properties (such as versions and build targets) are all pretty involved tasks, enough so (in some cases) warrant a build system that constructs the configuration for the main build system and so on. Include increasing compilation options, and it barely comes as a surprise that as time passes, the end output becomes as dependent on the set of scripts and rituals piecing it all together as it is on the actual source code. In terms of information, the build system (rather than some class design diagram) is a great target for inquiries on interdependencies and coupling between components but is also the first part of the chain to consider when ruling out suspects for some compilation error. As for intervention, most build systems of rank provide ways of replacing and redefining various pieces of source code right before they are being passed to a compiler or other tool. This is a dangerous but powerful ability.

Compiler

Compilers look at syntax and semantics in source code and are therefore great at telling you exactly where your input failed to oblige by some of its many rules. They are the ruling kings of static analysis with inside information on each and every structure of use in your code. It is the compiler that will

bind code to a particular machine ('target architecture') and in the process mess around with it in ways previously unthinkable. Code will get removed, replaced, reordered and injected during this process, mostly based on your coarse-grained preference.

When working with the compiler as a source of information on the composition of a system, it is of importance that each and every argument is both considered and understood in great detail, because even the most subtle of changes to the list of arguments passed to the compiler from the build system is likely to have a large impact on the resulting output[17]. Many subtle and domain-specific workarounds for various problems (including temporary fixes for problems with the compiler itself) are only ever visible at this stage.

A compiler can however forcefully be used for learning a lot about your code that otherwise would have been kept in the dark. A good example is the C preprocessor which has been the cause of a lot of developer distress. With the proper command-line arguments a good compiler can be set to storing intermediate representations, such as the code after a preprocessor run, but also the output before running it through an assembler or invoking the linker. Other potentially interesting pieces of information that can be obtained from the compiler are call graphs (both intra- and interdependencies, finer detail compared to the information from build systems) and actual sizes of structures (deviation between perceived structure size when looking at C code and actual structure size when padding and alignment are considered).

Last and possibly most important in regard to the subject of possible interventions, compilers can inject the data needed for outputting binaries particularly suitable for debugging or profiling.[18]

Linker (and Loader)

While the compiler is very capable of figuring out both how to tie a description to fit a particular hardware architecture and how the pieces are supposed to look and fit together, it is not particularly suitable for actually piecing all of them together in order to form a puzzle and therefore cannot reasonably be expected to fit this puzzle into the confines of an execution environment. Although the linker may be two-staged (one that links together into an executable, and one that on demand patches code and addresses at run time) the available intervention is similar enough that we can bundle them together as one and the same here. The difference (except the obvious one of build/runtime) is how intrusive the intervention can be. Patching several function calls by injecting into the final executable increases the requirements the particular executable may have on its environment in order to function and will affect

[17] One of the more popular compiler suites accepts some 1200 different command-line switches!
[18] For details on how this is done, please refer to Section 4.2.

subsequent tools (e.g., binary signing, checksumming) possibly making the executable(s) harder to instantiate on similar environments (for example, if you would like to run several instances of a debug-tuned executables on slightly different environments when trying to increase repeatability).

3.4.2 Miscellaneous Linker Features

A modern linker, together with a good object code format, provides many more hooks which can come in handy when diagnosing a system. It is not our goal to provide an exhaustive or even extensive list of such features; but you will have to go spelunking through your linker documentation on your own. We do, however, give a short list of three pet features we have found to be particularly useful.

- *Weak Symbols.* In some object code formats (most notably ELF), it is possible to define symbols as *weak*. A weak symbol will be used in the linking process only if the linker cannot find a better (i.e., non-weak) symbol with the same name. This allows developers of various library / utility functions to define a generic version of a function which can easily be overridden with a specialized version without patching the linker configuration.
- *Constructors.* Sometimes it is possible to mark an arbitrary symbol as a *constructor*, which makes the runtime linker execute that function prior to the `main` function. This is extremely useful for injecting more code from a separate file into an executable just as it is about to begin its execution. For all systems that support the linking of `C++`, some form of constructor support is necessary, but is it not always this convenient.
- *MAP files.* Many linkers can produce a text file containing information about the address and size for each object in the final executable. This can provide a very valuable help when diagnosing address-related problems during execution.

Functions by Proximity

When the linker combines executable entities from different object files, these are typically combined without changing the internal symbol order inside each object file. For example, if three files are combined the linker can place them in any order inside the final executable (such as {a, b, c}, {b, a, c} or {c, a, b}), but the internal order of objects inside each of the files a, b and c is typically left intact. The important consequence of this is that the original structure decided by the compiler is still present to a very large degree in the final executable file. In many cases, the order of objects used by the compiler is *the same as in the high-level source code*.

When analyzing a piece of machine code which is hard to identify, its origin can often be identified by considering adjacent symbols. This technique is useful when deciphering compiler-generated stub code, etc., which then can be traced to the original object file and then back to the high-level source.

Executable

The executable is the result of a long and hard journey. Compare the contents of source code to the end executable and it is evident just how big of a change the toolchain can make; symbol names have been changed and mangled, evaluation expressions re-ordered and possibly eliminated and lots more. To put things in perspective, this transition still pales before that of execution. In short, the executable is all the object pieces from compilation mashed into one binary blob together with some metainformation. The internal structure of an executable is dictated by a file format, which should contain all the information necessary to be able to instantiate the executable into a running program. As the executable is, in theory, the system condensed into a neat and tightly integrated package, it should also be a primary source of information about the system. Unfortunately, most such information is in a format that is unintelligible without considerable tool support and even then a lot of informal markers of no practical use to the machine or environment have already been lost. Although it can be considered an exception rather than the norm, there are situations where you need to consider both intervening with and gathering information from the executable:

- Broken Toolchain – With a broken toolchain the executable it outputs is in some way incomplete or flawed. Due to the number of tools and especially the complexity of some of the transformations performed on the original source code, it is fairly likely that certain combinations fail silently but yield partially garbled output. This is of course a highly undesirable situation but not an unlikely one when several binary protection tools are involved or when any of the tools involved are misconfigured in some way.
- Failure to Execute – A highly likely effect of a broken toolchain would be an executable that fails to execute. However, even with a seemingly correct toolchain the executable may be unusable if the targeted hardware architecture and execution environment is different from the one where execution is attempted. Problems like this one can be very subtle, as in the case of misalignment.
- Missing Primitives – Chances are that you do not have access to all source code and build-scripts, especially when dealing with legacy code and third-party components that were only supplied as object files or libraries. If a problem can be correlated to either one of these, one might have to enter the realms of *reverse engineering* in order to gain enough

information to either patch or replace. This can be considered a subset of the overreaching system analysis task but with a twist. Success might depend on looking at both static and dynamic aspects of the system. However, you have to start somewhere. With the aforementioned particular restrictions, the executable might be your best bet.

These are all situations which warrants instrumenting the executable in its static form. The main tools for this purpose are (in descending order of task complexity) *decompilers*, *disassemblers*, *object dumpers* and *hex-editors*.

Note that this design-write-compile-run flow for constructing software deviates slightly when considering interpreters (an ambiguous term but meaning a subset of virtual machines where the internal structures used in compilation are directly executed instead of being used for outputting code in some intermediate format).

All in all, these information sources are very convenient, due in part to their static nature but also because they can be distinguished and isolated with highly repeatable output. But yet again, they cover how the system was constructed with suggestions and restrictions on how it should be integrated into an environment, not how the system will actually operate.

3.4.3 In-Execution

It's been established that software is a fast moving, shape-shifting target. During execution, chances are that the window of opportunity is tiny and that the possibility of successfully picking the right time and place for instrumentation in a coordinated fashion will be distant. Due to the tight intradependencies of software already at a microscopic level, we might also actively trigger, or inadvertently tamper with its own reconfiguration. This problem is nothing new. It's been discussed time and again as *observer-effect sensitivity*. The problem is that we can't simply describe it and then hope that it will go away but are rather forced to take it into consideration. With pre- and post- execution, gathering data isn't much of an issue, but pruning/discarding everything that is irrelevant to a specific case is, however, is somewhat more problematic. With execution we still have that problem (at an ever larger magnitude) but also the additional problem of actually getting hold of data in the first place. Now, the available targets for sampling depend to a high degree on the amount of control or influence you have at your disposal, as there's most probably some protective mechanisms in play (a simple example of such a mechanism is process isolation in terms of execution and memory, which is supervised by most current operating systems). At the very least you ought to be able to provide the system with some input and obtain some or all of its output but ultimately, the precision of your runtime analysis will vary based on the amount of control you are able to exert.

The ideal situation in this regard would be to not only be able to read/write all of both code and data but also to skew timing and modify external dependencies, something mostly possible with absolute control of the enabling machinery. With such control a range of options reveal themselves: full system virtualization, cloning, emulation, etc. Even regular source-level debuggers (the kind integrated into most IDEs) rely on a fair amount of control, in reality only possible through close cooperation with underlying operating system and support from hardware. Additionally, relying on the notion that you actually possess such control might prove somewhat deceiving as it is to a fair degree not possible[Garo7]; the control mechanisms in play are well-known (which is something to take into account with systems that have additional shady forms of protection[Biono6], but your mileage might vary.

When forced to act from the perspective of severe limitations in terms of control, there's also the techniques of the hacker community to consider. Many software bugs have consequences for software security, meaning vulnerabilities. A vulnerability that can be exploited is serious enough, and while there are situations that warrant ethical consideration, the means for exploitation are typically not that difficult to grasp and use by whatever means necessary. After all, they are means for gaining some control of a subject (or elevate the existing one) that differs quite a lot from the regular ones (breakpoints, etc.) with an almost disturbingly rapid growing set of available tools far ahead of most other in-execution instrumentation endeavors.

Situations that call for in-execution analysis vary a bit more than, for example, crash dump analysis and it is therefore somewhat harder to generalize and discuss various ways of measuring, representing and intervening thus requires a bit more in terms of individual ability. Let's look at a few ways of treading lightly through dynamic information sources (and vantage points).

External Connections, I/O

Simplifying things a bit, let's say that software either processes data (automated calculation), responds to user interaction (interfacing representations) or both processing and responding intermittently. The latter two cases imply that some channels for passing information in and out of the system are required and that they can be coarsely controlled by interception, altering data flows with respect to boundaries, type and protocol. This is something that *fault injections* and similar techniques have employed for a while. As intentions here are somewhat more refined than 'making things go boom', one ought to act in a more informed manner, but the approach in itself is not all bad with the mindset that you are interested in figuring out the contents of the black box rather than blindly accepting that the box is responsible for things beyond your comprehension. Some prerequisites for not just spending time acting out the role of a poor random number generator would be the ability to gradually trace execution and data flow and their correlation (or lack thereof) to the

originating event, but implies the need to either step through execution or roll-back to a point prior to insertion (or, with interface that is repeatedly invoked, look for traces of feedback loops). Examples of such connections would be OS enforced message/event loops and File-/Socket- I/O operations but could be any kind of known, reachable entry point.

Entry points

The number of software components in development or deployed that lack the notion of third-party libraries, modularity or even system calls are rapidly moving towards extinction. This means that we are, for all intents and purposes, not blindly cutting into software at some random offset, but that there's lots of entry points that can quite easily be figured out. An entry point in this context means both the implementation of an interface (either place where it's being called: at a trampoline or at the particular function prologue) but need not be external per se. A prerequisite and identifier is that the code it is coupled to adheres to some kind of known convention or protocol, which implies that we know (to some degree at least) about the assumptions at a level of detail corresponding to our incision (often on an instruction-by-instruction basis). The underlying reason for this is that we in turn can assume that at least there is some kind of limit to the potential backlash that might arise from intervening without a complete picture of the causal factors in play. Entry points devised from symbol tables, linker maps, instruction pointer sampling, system calls, instruction idioms, etc., are all easy targets for instrumentation, as the activation of any injected probe tells not only where execution is occurring but also when it is occurring. By cross-referencing with point-specific data and the results from other proximate probes it is possible to get an idea of why execution reached this point at a fairly fine-grained level. The tasks them become about determining if this is desired and expected or not.

Contrast Material

The notion of various connections to surrounding environments, as well as internal bindings and correlation of entry point specific data shows that there's also a point to understanding data in its various forms instead of only considering execution. It has already been discussed as to how data can be depicted for a biased form of interpretation, which is the idea of representations. A somewhat complicated area in this regard is that of types: abstract types, aggregated types, hierarchies of types and so on.

A machine will only enforce the integrity of data of some type at the level at which it operates. A `CPU` for instance has an implicit sort of guarantee that some rules of manipulation of the contents of registers will be enforced. Similarly, a memory controller makes sure that the contents of a particular location will be retrieved, stored and updated according to another set of rules. This is

pretty elementary. The twist is (and does hopefully not come as a surprise to anyone) that the integrity of less basic types, like control information used to manage instances of objects in some object-oriented environment or at a lower level – for example the contents of a C struct, is not guaranteed by the machine (as user-defined types were unknown at the time of machine construction). Therefore, various means necessary for supporting the runtime management of language-specific data & type integrity are inserted into the generated output at compile time. These support-systems can be made to be rather complicated (and moving back from abstract to primitive again) like in the case of Virtual Machines, but note that any problem that affects the integrity of data and type (which is basically a fancy way of saying *data corruption*) propagates outwards, determining if the data gathered from some entry points a few virtualizations down the road is intact or not, can be a challenging task.

To assist in this regard, we can apply contrast material in its various forms. Since we know some things about the system at this stage (from external connections and I/O to entry points), it should be possible to inject known data at one of these established borders and follow its track as it passes through the system. To increase the chance of success (and avoid an immediate crash), the data can be safeguarded using techniques such as canary values and checksums and choosing borders that accept data formatted in ways with *flexible* restrictions in terms of size and contents. Formats like those from markup languages are obvious candidates. This has been used to great effect in a lot of different contexts, from testing/security as *fuzzing* to security/exploitation as *heap spraying* [Sotirov08] but should ideally be used in a more informed manner.

Anatomic Components

A large problem with diagnosing and fixing software when compared to other disciplines, whether it be medicine or plumbing, is that software is a domain where it's hard to find useful patterns for guided and precise troubleshooting. In all of its essence, the smallest building blocks here (opcodes and operands) are well-known, and at this level it is cumbersome yet possible to extrapolate patterns for identification (idioms) although these do not correspond well to classifications of problems and their sources. With software, you seldom have the luxury of dealing with abstract-reasoning on a single level or even a few, but you are rather forced to deal with them on several levels at once. Dealing with this and coming out victoriously is something that signifies a disciplined and well-versed programmer. Connecting all abstractions involved together to form some sort of generic anatomy for all software seems, unfortunately, rather futile. Does a web browser have something in common with a database server? Does the file storage of two different database servers have something in common? It would be foolish to say that they do not, but actually finding ones useful enough from which to generalize in order to make an anatomic model may prove a bit more difficult. The key seems to be finding out the parts that

actually are similar on the level of mechanism rather than perceived function or utility. Luckily, the few that are to be found are also already conveniently placed in standard libraries of programming languages or language features by themselves. The most evident ones would be the memory allocator/deallocator/manager, userspace thread scheduler, string manipulation functions and regular expression engine. An example of an interesting pattern using memory management as an anatomic component would be something akin to `C malloc` returning `NULL`, which can lead to quite interesting complications[Dowdo8].

The benefit of these *anatomical components* is that their interfacing and behavior together with their problems, tend to be well-known and their function is also employed by other software[19], so that usage patterns can be compared to ones employed from using the same component in different software.

For further discussion on in-execution data sources and how they fit together, refer to the section on *Analysis Imperatives*(3.6).

3.4.4 Post-Execution

Something failed to go as planned and, all we have left are the smoldering remains of the once proud executing system, remains that take the shape of some binary blob: a snapshot of as much of the system state that could be piled up and tucked away right after the moment when everything came crashing down. Either the enabling machinery's built-in protection decided to pull the plug on the process or the software terminated itself due to some safeguard-like error handling code.

This binary blob is commonly called a *core dump*, a low-level version of which will contain register states, stack and possibly heap contents, process information on open file handles, trace of recent system calls – basically whatever metadata that can reasonably be squeezed from supporting machinery. In the case of any run of the mill desktop PC software, this is about as far as you will get. If you are dealing with some in-house developed embedded solution, chances are that you have access to more detailed specifics about what various system states represent, which can be used as a sort of structural overlay to make some parts of the dump more intelligible.

Crash dumps, in short, condense all post-execution information of relevance and are often actually the first signs of bugs in unstable systems. They crash so often that more complicated and involved problems never get the chance to appear. So, in essence, a crash dump is both *the one* and *only* source of information in the post-execution systemic state.

Post-execution analysis may be one of the most beneficial sources of information given some fairly common circumstances, and for many situations, it is

[19]Be careful about versioning in this regard.

the only one available. Here follows a short summary of conditions that make post-execution analysis especially interesting[20]:

- Low-repeatability – A crash dump from a system where the crash isn't guaranteed (the case with many rather complex underlying issues, especially when dealing with concurrency) may be rare evidence that should be taken into special consideration. Before discarding any crash dump as uninteresting or *too broken for words*, make sure that the problem is highly repeatable, otherwise the dump should be tagged and stored for special safe-keeping.

- Highly instantiated – While there surely are many systems tailored to fit some very specific end user need targeting a low number of customers, the case might still be the opposite: thousands or millions of instances made from the same build executing in slightly different environments. Combined with other mentioned conditions, a high amount of reports gathered from a large control group automatically compared to other dumps to assist in pruning out irrelevant information or the inverse, highlighting likely suspects, makes crash dumps a good and perhaps exclusive, source of valuable information.

- Deployed software – (might be similar to repeatability, but not necessarily) Lots of software displays quite remarkably different behavior when deployed in some setting outside of the regular development, experiment or testing environment. Problems that you were never before forced to deal with during development suddenly starts to occur during the most delicate of circumstances, situations where large sums of money and industrial reputation may be at risk. Any way of retaining and collecting at least some crash-state information may prove key in solving sensitive problems where margins are running thin and stakes are growing high. Larger deployed software (operating system suites, web browsers, etc.) are early adopters of this practice by having a crash catching fallback plan requesting that the user allow it to pass crash information on to developers.

- Specialized system – In contrast to the office suite or similar end user software, troublesome code may be deployed for systems where the developer possesses a larger degree of control over environment parameters than normal, this is especially true for the embedded kind of devices. In these cases there should already exist pretty well-grounded models of data formats, memory address mappings and so on, providing a more potent base for finding clues, in order to make better sense out of a large and otherwise unmanageable amount of data.

[20] Note that all primary concerns here revolve around repeatability in one form or another.

3.5 The Software Analysis Conundrum

Sources of information, especially in-execution and post-execution, have some underlying problems shared with our analysis approach in general that make matters more complicated and warrants special consideration on its own. This section will be used to highlight these problems in a backward chaining fashion using the post-execution systemic state as a starting point, to show the fallacies in the way some reason about software systems.

In terms of terminology, let's make a few distinctions. With *post-mortem* we refer to everything we can ascertain from the fact that a system has crashed. The largest part of *post-mortem* is *crash dump analysis*. As contrast, we have *post-execution*: a state where the system may or may not have crashed – the only certain thing is that the system is no longer executing. The difference then between *post-mortem* and *post-execution* is that in the *post-execution* state, system execution can be resumed. In some odd cases, like when *the system* targeted for analysis is software managed by a multitasking kernel, the post-execution state is intermittently recurring. If the state space covered by the system then gets serialized, we get a *snapshot* or an *image*, rather than a *core dump*[21].

Snapshots and crash dumps are the central sources for information at a post-execution systemic state. They spring into place from different, but similar mechanisms. A snapshot is generated upon some external request whereas a crash dump is generated as a reaction to some event that renders the system terminal, the overarching process being analogous to that of a human autopsy. Common conditions that precede a crash dump (e.g., machine check exceptions, illegal instructions, watchdog events) or attempts at overstepping boundary conditions (memory alignment, page protection, memory addresses, etc.). This difference implies something about what the dumps contain, namely the aftermath of the particular condition.

If the system is damaged to the degree that it crashes, chances are high that there may be a *proximate-onset-proximate-cause* type scenario in play; close to the crash point (at a low level, code pointed to by the program counter), there will at the very least be signs of the immediate cause of the crash and by backtracking changes (at a low level, unwinding stack) to the relevant state and locations from where these changes were initiated, it is likely that the culprit will be within the reach of a few instructions. A good debugger will have direct support for reading core dumps, automatically mapping addresses to symbol names, instruction pointer to respective lines in source code and similar functions.

Let's go through a little modeling exercise to give this some perspective. Consider Figure 3.10, our naïve view. The executable resulting from building some

[21] Although in the wider sense of the word, a core dump can refer to either a snapshot or a crash dump. The use of core dump as a way of referring to a snapshot taken after some undesired event is here considered being the norm.

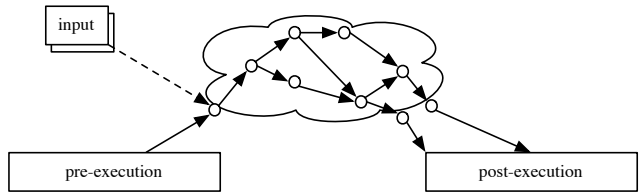

Figure 3.10: System view, naïve

system is set to run on a machine. Execution (the arrows in the cloud) flows from one function to another, with the output of one being fed as input into the next. The first function is fed by a set of inputs that have either defined statically or are the result of the output of some other system. Branching may occur, but it only adds some minor complexity to the mix. Eventually execution is halted, we determine if the output matched expectations (correct execution), if the output mismatched expectation (anomalous execution, time to fire up the debugger) or if a crash condition was triggered and we have a dump on our hands.

Now this model is a bit too exclusive for our ends, so let's expand upon it. Looking at Figure 3.11; we consider the cloud-like thing around the arrows to now represent the state space within the (epistemic) confines of the system. Each function may now affect not only the succeeding function but also an internally shared state space (registers, memory, etc.), a space we have previously described as huge[22]. Furthermore, we accept the notion that our system is not isolated and could not be successfully modeled as such. Therefore, we add another box outside that denotes the external state / runtime state. For a software where the execution is aided by an operating system, one example of an external state would be a file handle. Additionally, the set of inputs is not only used as a function argument, but is merged into the shared state space.

This model is still a bit too exclusive, so let's add another cloud with terrifying consequence. In Figure 3.12, we add the notion of state persistence, which means that the effects of execution are not only being fed back into the executing system but potentially into subsequent runs of the same code. Finally, this persistent state is not exclusive to the particular system but is potentially shared by several and with intersections to parts of our external state. Expanding the file handle example of the external state we now also have the file system to which the file handle (amongst other things) acts as reference. There are several other large parts not covered by this model (like limitations in time and space imposed by the shared resources), but it is sufficient to cover

[22]Research in formal verification like model checking refers to the exponentially sized state space in relation to a system description (source code/compiled instructions) as the state explosion problem.

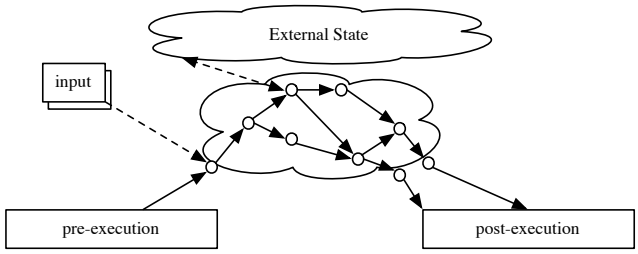

Figure 3.11: System view, cynic

relevant aspects of post-execution analysis. Comparing the progression of this model to the one of computing described in the *Software and Software-Intensive Systems* section of the Introductory chapter, we have walked along a similar path, from punch cards to SiS but still not transitioned into the complexities of Software-Intensive Systems either in terms of heterogeneity, distribution or concurrency.

Not refining the model further, what can be deduced about crash dumps and system snapshots based on what is currently covered? If we start with the naïve view of Figure 3.10 and assume that one of the two arrows leading to the post-execution systemic state is a transition made due to some panic condition, i.e., a crash, it seems completely feasible to invert the entire scheme and traverse back to the state where execution took that unwanted turn. Reasons for this are two-fold, but they are somewhat related. In part, there's some clear shortcomings in the precision of the model. All kinds of states are aggregated into a big mystical space where there is no distinction made between the software and the machine. In reality, the state space for a processor is connected to the state space for some piece of software, but when reasoning in this context it is reasonable to accept some abstractions. Secondly we assume that each and every function here is invertible and/or reversible corresponding to a currently unresolved fundamental problem in computer science: *Do one-way functions exist?*, if they do, we cannot directly start from the point of a crash and change gear into reverse. Looking at a more practical standpoint, reversing execution at a machine code level may be possible for many instructions but becomes problematic already at conditional branching. Additionally, the quickest of operations are done on registers, of which there are quite few available, but still act as the primary intermediate containers for state. Due in part to their relatively small numbers, the state they contain will be overwritten several times during the course of a function. Therefore many intermediates states necessary in order to reverse execution, are lost as illustrated in Figure 3.13. Without additional support, the actual window of opportunity that

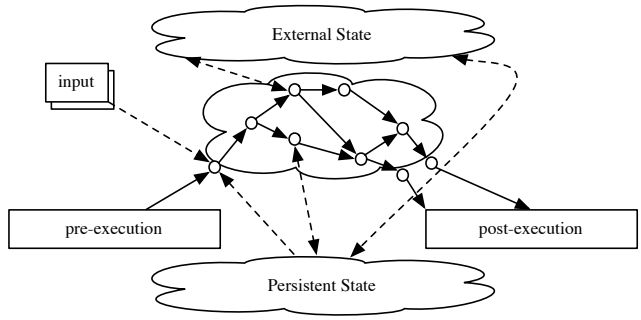

Figure 3.12: System view, bitter

can reasonably be exploited to reverse execution is tiny[23].

But there are other possibilities within the model. Use the information of a crash dump to devise the conditions that triggered the crash (as the immediate cause is closeby). Then restart the system, and at the point of each completed function check if we're approaching the conditions that triggered said crash. This way we get to explore system states before the effects of a problem starts distorting the state. Such distortion is unwanted as it reduces the value of information that a crash dump could provide. Now we can generate a new dump that should be at least one step closer to the ultimate cause. Ideally, the immediate and ultimate causes are the same here, and a single crash dump suffices to tell us what and where to fix. As the harsh reality seldom has room for ideals, we will eventually have to work on cases where the immediate and ultimate causes are distant from one another. This adds a lot of tedium to the process, but our problem has primarily turned into a state space search, one which to a good degree can be automated.

The underlying issue with this approach (otherwise we'd make everything crash, work with the problem as just stated and debugging would be far less of a challenge for the majority of bugs) is that of repeatability and state explosion. We presumed that the problem would be completely repeatable, which within this model is plausible as there is little to suggest otherwise. State explosion fundamentals have already been covered and apply here as-well. With an exponentially growing search space, our search has to be pretty clever (meaning complicated heuristics) or our problems reducible to very small portions, small even by academic standards.

[23]For specific targets when dealing with embedded and small systems, there are third-party debugging solutions that do provide the ability to rewind execution by up to a few seconds.

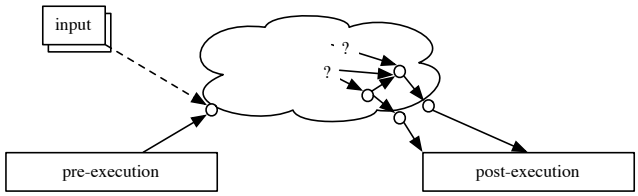

Figure 3.13: Distorted naïve system

Throughout this chain of thought, there's been the underlying assumptions that a) by using the input of a core dump from a crashed system we can deduce what the immediate cause was (as shown, we can!), b) we can restart (or rather, create a new instance of the same system) and determine that we're about to step into the condition from the first point listed (we can!) and draw the conclusion that this could be repeated iteratively until we reach the ultimate cause. The problem is that the time from a to b may be very long and that the same goes for the full chain from immediate to ultimate. The steps described imply that some system execution is performed each and every time that can be assumed to be working as desired (otherwise, there might be a point to recreating the solution in its entirety and firing some developers in the process). By using mechanics similar to a core dump, we create a snapshot of the system at some point of execution (say halfway from start to crash) and instead of re-instantiating the system for each run, revert to the snapshot. This is a strategy that can be performed iteratively as well and can reduce all time consuming factors considerably.

Unfortunately, this model validates poorly to software from the illusory *back in the day* era and even worse when compared to to the current state of affairs. Thus, we move onto the refined model, shown in Figure 3.11. Here we introduce the notion of external or shared state and the scope of problems expand exponentially. This implies that there's other factors that we may have to consider foreign to our own. The consequences of such an alteration are pretty far-reaching in the sense[24] that any state may suddenly be influenced by an external source, and subsequently *pollute* the system further [Chow04]. This also increases complications of post-execution analysis as external factors also have to be taken into consideration. The idea put forward a few paragraphs ago, i.e., that we could reverse execution given some properties of operators and state space is now even more distant. In addition, the shared state space further complicates the notion of repeating the execution of a previously instanced software as we now also need to recreate similar conditions in both space and time for any and all systems working with the shared state, al-

[24]Recall that we're still considering things from a post-execution perspective.

ternatively record/replay requests to and from external state which can only reasonable be done in a highly virtualized/controlled environment. To exemplify somewhat, many operations on file system entries require that you pass some sort of identifier tag along as an argument to the function call. The process allocated memory that stores the value of such an identifier tag is by itself a state holder, but also represents an external connection into a neighboring component like the OS kernel which also maintains some state on behalf of both the particular processes, other processes utilizing the same resource and for the kernel itself. The point of the matter for this version of the model is that in order to apply the steps from the previous one when reasoning from a crash back through the causal chain on a system with external state modifiers, control will also have to be expanded outwards to cover all processes involved.

Ultimately, not even this model validates properly to current software at the level of abstraction it targets, thus further expansion is needed. Therefore, we look at Figure 3.12 for the last magical symbol that should set things straight. The persistent state then simply covers whatever possible stateholders that will retain their state after the system has finished execution. These are by nature also external and may be shared but the effect they impose that makes distinction interesting is that they add to feedback loops for subsequent runs of the same system. It then follows that what may lead to a crash in one run of the system may be gone or changed in a previous one. To complicate things further, persistent state holders may well also be modified by other systems not only during execution but also between runs.

3.6 Analysis Imperatives

The analyzing of a system is a cooperation between several important parties, most notably *the analyst* and *the target system being analyzed*. In some cases the analyst can modify the target in ways other than merely simple intervening, for example, by making special builds for a customer, extended testing or placing requirements on future software versions.

Given the possibility to affect a system to ease future analysis,[25] we would argue for three points: make the system *fail early*, *fail often* and *fail hard*.

3.6.1 Fail Early

The longer it takes from something failing to the failure being discovered the harder it is to find out what happened. As time elapses important evidence from the system is lost, but also the people working with the particular something that failed forget about the assumptions they made and their possible

[25] We imagine this would be called debugability in the software engineering lingo du jour.

implications. Minimizing the time from failure to discovery is important for many types of systems, software intensive systems being no exception.

What is special with software systems is that we can express this imperative in a different form that is more operationalizable: *avoid execution as soon as possible*. While execution is where the action is, it is also a place that is comparably hard to analyze and where a system easily can damage its own state. Thus, in order to fail early and permit easier analysis we want as little execution in the system as possible, something that can be applied to the system in pre-execution and in-execution states, as is illustrated in Figure 3.14.

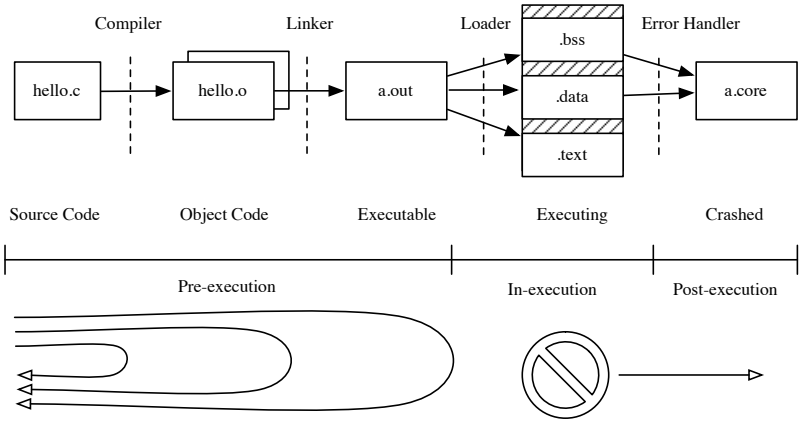

Figure 3.14: Failing early means avoiding execution as soon as possible

Pre-execution

Certain problems, which are likely to cause unwanted behavior, are visible already in source, object or binary code. Preferably, such issues should be automatically identified by the toolchain as it transforms the software. In the simplest case, the original developer is made aware of the issue and fixes it thus avoiding unwanted behavior at all.

Generally speaking, errors and warnings reported by components early in the toolchain are easier to interpret than their counterparts reported late in the chain. An error or warning message reported by the compiler is trivial to understand[26]. Messages from the linker are significantly harder to interpret, and problems reported from the loader may require some thought to interpret.

[26]Some messages from a C++ compiler contradicts this.

When analysis at the systemic level is required, we assume software has already successfully been transformed into an executing state and that there are no obvious errors to be found by the toolchain. If the analyst expects certain issues to cause unwanted behavior, he may intervene with the toolchain or source code to make the transformation more restrictive. This allows him to rest comfortably in his armchair as the compiler or some other tool processes the source code looking for the particular issue at hand. An example of this approach is illustrated in Figure 3.15 where a single `#define` causes a conformant C99 compiler to reject non-constant formatting arguments to `printf`[27].

```
1  #include <stdio.h>
2
3  #define printf( ... ) printf( "" __VA_ARGS__ )
4
5  int main( void ) {
6    char * fmt = "%s\n";
7
8    printf( "%s\n", "hello, world" ); /* Will compile OK */
9
10   printf( fmt, "hello, world" ); /* Causes compiler error */
11 }
```

Figure 3.15: Line #3 makes the compiler reject non-constant format arguments.

In-execution

Software executes at a tremendous speed; state changes, data moves around and there is no way for a human analyst to observe everything that goes on. When things go wrong, the analyst wants to know *what went wrong*, typically at a very low level of abstraction. In order to find this out, it is important that the software does not continue to execute, as doing so will change the state and move data around thus making it harder to find out what actually went wrong with the software.

Whenever executing software encounters a runtime condition from which it cannot recover, it should stop execution, the sooner the better. A good example is the use of `assert` functionality where a program can assert that a particular internal state is valid before taking some action[28]. By asserting the valid internal state, the software does not continue execution into domains known to be bad.

[27] We leave it as an exercise for the reader to figure out why one would want to do this.

[28] Assert functionality is, unfortunately, also severely misused by some developers. Software should assert that its internal state (to a function, class or similar) is valid. It should not assert the values of parameters, etc.

When using `assert` functionality, the system should halt *as soon as possible*. Even the simplest code in an assert or error handler can easily alter important state which would otherwise be usable by an analyst. An assert function should preferably make the machine crash immediately by throwing an uncatchable exception or similar.

```
#include <stdio.h>
#include <stdlib.h>

/*
 * Attempt 1: Naive
 */
#undef assert
#define assert(x) if( !(x) ) exit(1)

/*
 * Attempt 2: High-level
 */
#undef assert
#define assert(x) if( !(x) ) do { \
  fprintf( stderr, "Bad␣assert␣in␣func␣%s␣on␣%s:%d\n", \
    __func__, __FILE__ , __LINE__ ); exit(1); }while(0)

/*
 * Attempt 3: Portable low-level
 */
#undef assert
#define assert(x) if( !(x) ) abort();

/*
 * Attempt 4: Machine-level exact
 */
#undef assert
#define assert(x) if( !(x) ) asm( "int␣$3" )

struct sample {
  char *tag;
  unsigned min, max;
};

static void prepare_sample( struct sample *s ) {
  assert(s && s->tag && s->max >= s->min );
  /* ... */
}
```

Figure 3.16: Please assert that no external state is damaged

As an example of how good and bad assert handlers can be implemented in the C language, consider the source code example illustrated in Figure 3.16. In the first, naïve implementation the program simply exits whenever some unacceptable condition occurs. There is no way for the analyst to tell which assert failed, and all internal state in the program is lost when it exists. The

second, high-level implementation prints a diagnostic message with the name of the function as well as source code file and line. While extremely useful compared to the first version, there is no way for the analyst to tell which of the prerequisites to assert that actually failed – was the pointer `NULL`, or `min > max`? Also, as the software just exits, important state that could tell what actually happened is lost.

The third, portable low-level implementation calls the system function `abort`, which creates a core dump of the system. This dump can later be analyzed using a post mortem tool to determine not only which assert failed, but also the runtime state that caused the assert to fire. The problem with this implementation is that *the function call to assert may damage important state*. There is a risk that some subtle elements of state, such as the value of `CPU` registers and memory management unit (`MMU`) configuration, are modified as the function is called. For a high-level system this may be acceptable, but for an embedded target one would rather aim for a lower-level solution that *does not destroy state whatsoever*.

As example of such an approach, consider the last attempt, the machine-level exact, where a special instruction to the machine is used to cause a crash. It should be noted that this example is not directly portable across different machines (the `int 3` is for X86), but we have not yet seen a `CPU` that has lacked hardware support for a clean crash.

When targeting a virtual machine there is likely a similar function available, often with more bells and whistles. Irrespective of the primitive used, make sure the system *aborts execution as soon as possible and without damaging state as soon as an invalid state is discovered*.

Special Case: Programs Processing Persistent State

A special case of *failing early* concerns software that writes persistent state, for example by serializing to a database. Make sure such software asserts the internal data state *both when writing and reading data*. It is hard to analyze a system which only validates integrity when reading data but potentially writes damaged data for later use. Preferably, when serializing data some form of version tag that identifies the software build should be included in debug versions to assist in tracking down problems relating to data format errors.

3.6.2 Fail Often and Fail Hard

Previously, we have discussed our inherent inability to create bug-free, perfect software systems and accepted the notion that all non-toy systems have had bugs in the past, have bugs today, and will continue to have bugs tomorrow and for the forseeable future. Also, we've learned that the analysis of a large software system is anything but trivial, and analyzing a complex bug may very

well require weeks of work from a team of skilled analysts. Thus, the question arises: *Which bugs must be fixed today, which can be left for tomorrow and which can we simply ignore for now?*

Analyzing a bug is typically far more time consuming than fixing it. Thus, the results from an analysis are a poor input when deciding what to fix and what to ignore[29]. To make a good technical decision on what to fix[30] and what to ignore, several aspects, both quantitative and qualitative, must be considered.

The main quantitative input is *the number of times a particular unwanted behavior has occurred*, and the main qualitative is the *the approximate damage each such behavior causes*. The latter requires some form of pre-analysis but not an explanation of the causal factors behind the behavior.

Obtaining reliable information about the frequency of a particular unwanted behavior is trickier than it first seems. Traditional testing can reveal some figures, but it is hard to know if these are representative of typical system use. Compared to typical use, testing finds too many issues that are triggered with a short state (e.g., clicking rapidly on this button makes the software crash) and too few with a complex state (file system inevitably crashes with out of disk after one year of use).

Describing unwanted software behavior in a precise and understandable way is not a trivial task. This is painfully apparent to each and every analyst who has tried to decipher user-written defect reports. While perhaps not all reports would qualify as creative writing, many of them are completely useless from an analysis perspective. Thus, even if we (probably incorrectly) assume that users actually write defect reports proportional to the number of times a particular issue has been observed, trying to figure out which of these reports concern the same behavior, not to mention the same underlying issue, is never-ending work for which few analysts find time.

So, rather than relying on testers and users to provide relevant data about testing and usage, we'll have to rely on the software system itself, for example by making sure it *fails often* and *fails hard*.

Often

Should a subsystem crash or otherwise behave in an obviously incorrect way such as violating protocol, the system analysts want to know about this. Fortunately, this is apparent to the executing software system: some error handler will be invoked on a crash and adjacent components will notify violation of protocol. In all these cases, the software subsystem should *fail*, which, for buggy systems, means that the software will indeed *fail often*. Also, as much

[29] Also, there are often both political and technical aspects to what should be fixed and what should be ignored, but for this discussion we consider only the technical.

[30] Actually to decide what to analyze with the goal of fixing.

state as reasonable from the failing subsystem should be collected and attached to an automatically generated error report.

Systems connected to the Internet often use some software agent to transfer these automatically generated error reports. This allows direct feedback from the server to the user if a newer version of the software exists and corrects the particular issue. Systems that are not connected to the Internet or that are subject to restrictive rules that prohibit external communication cannot as easily transmit state but should, if possible, save state locally to allow some form of later retrieval.

By failing often the software system emits not only information about which unwanted behaviors are more common than others, but also provides state that significantly helps in later analysis of the bug.

Hard

Closely related to *failing early* to avoid damaging important state and *failing often* to get usable statistics is *failing hard* to make sure the few scarce clues that aid system analysis aren't lost in some log never to be read or even piped to `/dev/null` by a sinister system administrator.

If a situation occurs where a part of the system crashes, a serious assert fails, or if the system severely violates protocol, attention is needed. The brutal way to obtain such attention is to simply terminate the system and wait for upset users to call. While this is a suitable technique for early debug builds, it is not well-suited for so-called final versions used in a production environment. While political considerations may mandate the use of other methods than system termination, always *make sure information about system failures reaches the analysis department*.

References

[Gaines65] Gaines, R. S., "On the translation of machine language programs", Communications of the ACM, vol. 8, no. 12, Dec. 1965

[Biono6] Biondi, P. Desclaux, F., "Silver Needle in the Skype", Blackhat-EU '06 Conference presentation, 2006

[Sotirov08] Sotirov, A., Dowd, M., "Bypassing Browser Memory Protections", Blackhat 2008 Conference, 2008

[Chow04] Chow, J., Pfaff, B., Garfinkel, T., "Understanding data lifetime via whole system simulation", 13th USENIX Security Symposium, 2004

[Dowd08] Dowd, M., "Application-Specific Attacks: Leveraging the ActionScript Virtual Machine", IBM ISS Whitepaper, Apr. 2008

[Gar07] Garfinkel, T., Keith, A., Warfield, A., Franklin, J., "Compatibility is Not Transparency: VMM Detection Myths and Realities", Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI), 2007

4 Tools of the Trade

We have touched upon the idea of tools to automate and assist various parts of our software-related endeavors many times by now. Some of these have been described in fair detail, but the majority have up to this point been fairly abstract *instruments of thought*. In this chapter we will return to a more technical perspective.

Human beings in general are avid tool users and to a fair extent able to use tools without much knowledge or instruction in regards to their construction or even on which principal grounds (mechanics) they operate, through the use of repeated experimentation. Presumably by a similar underlying process, we refine (optimize) the way we use some tools to produce, for some purpose, more precise (better) results. By understanding the mechanics involved, we can both refine our tools but also put them to even better use.

The developer, just like other craftsmen, has an extensive array of tools at his disposal. In *Software Demystified*, the cooperation between tools (like compilers, linkers and loaders) that piece software together was covered to the point where a description had turned into an executing software, and now the time has come for the selection of tools specifically created for studying and tweaking running targets. Although knowledge on the usage of, for instance, debuggers is fairly well known, the mechanics on which they rely may be a bit more shrouded in mystery.

4.1 Layout of this Chapter

The following section will start with a generic, high-level view of a debugger described by the core functionality it implements and a little on the underlying support needed from hardware and operating environment. With that in place, various levels for implementing this functionality is reviewed. The section on debuggers will be closed with some discussion around the kind of problems such debuggers face. The sections on tracers and profilers are laid out in a similar fashion, going from description to details, but both sections build upon some of the details given in the Debugger section and should therefore not be considered as being independent.

4.2 Debugger

The term *Debugger* has unfortunately already turned into an ambiguous one. The most frequent use seems to be that of a source-level debugger, probably because it is one of the tools closest to the larger majority of developers; most larger IDEs and tool suites integrates one or several sorts of source-level debuggers.

On the obvious side, a debugger is a tool that helps with debugging, but this is not a useful distinction. To refine the definition somewhat, a debugger assist with debugging by allowing manipulation, the *intervention*, of execution flow and *measuring* of the state. The idea of source-level debugging denotes a form of *representation* where commands and measurements are mapped to source code. One important thing that can be deduced from this description is that *a debugger is a dynamic tool working on an executing system*.

Manipulating Execution State – This is a necessary feature for a debugger. As stated in the chapter entitled *Principal Debugging*, a large problem with a dynamic and executing target is the state and particularly the rate at which the state changes. *Where* you alter the execution state is not the only variable in the equation; *when* is involved also. At any rate, the typical[1] approach to manipulating execution state is through some collaboration with the enabling machinery where a location (memory address or similar) is set as a trigger when execution occurs or when information at the particular location is altered. The handler associated with the trigger uses some event handling mechanism (through an `OS` interface or `CPU` Interrupt Request(`IRQ`)) to communicate with the debugger.

Exploring the State – On the other side of the coin, there is state exploration. While merely altering execution state may have some merits as an *intervention*, it is better to attach some sort of meaning to the fact that an interruption in the execution flow has occurred. In the easiest and best of cases, the interruption occurred at the single instruction that when executed will transform the program from being in the correct state to an erroneous one. If that was the case, it would be possible to start investigating the state, backtracking the compilation toolchain back to the source level to see which programmatic construct was the culprit.

In reality, things are more complicated. One line of source code results in several instructions, with several execution paths being traversed in parallel on different `CPUs`, and the state space even for a simple few instructions can be huge. Therefore quickly pruning irrelevant data and presenting data in ways perceived as relevant (for the analyst), becomes important.

[1] There are literally hundreds of different kinds of debuggers, and state management is a primary parameter that varies between these.

From these two overarching tasks, we can extract some prerequisites:

- Intervene – Be able to access and alter various parts of the debugging target.
- Measure – Read values from the target state space to as large an extent as possible.
- Represent – Decode instructions (disassemble) and present data.
- Synchronize – Regulate when control is exerted to alter target execution flow.

Generally, the faster and more reliably these prerequisites can be fulfilled, the chance that tools built on top of such functionality can be of use increases. In addition, the more that can be moved to being part of the enabling machinery the better, both due to performance but more importantly in order to reduce potential observer effects or at least control such effects to a finer degree. Additional functionality that is by no means necessary but considered extremely helpful in terms of machine-level support is the ability to reverse execution flow on an instruction by instruction level of granularity.

4.2.1 Straight and to the point

A quick glimpse at the functionality provided by most kinds of debuggers, involves a variety of points with *breakpoints* and *watchpoints* being the most notable ones. In essence these are reference markers connected to some level of abstraction above raw machine code (from assembly-level instructions up to some line or variable at source code level) giving the debugger rough instruction on where instrumentation is to be done.

A typical high-level workflow involves the analyst singling out some region of code where runtime inspection is needed. Then, a breakpoint is placed pointing to the first possible opportunity in the target region. The program is run (or re-built with debugging information included) either directly by the debugger spawning it as a child process or by the debugger latching on to the process upon execution. A breakpoint is inserted at some viable address close to the intended destination. Eventually execution hits the breakpoint and is redirected to the debugger which notifies the analyst that execution has halted. At this point a variety of information sources is accessible, such as the values of current variables, current stack contents and outputs from a call trace. Furthermore, several flow control options are enabled, such as the ability to step forward to the next instruction (or line of code), step into into a particular function or skip some executions altogether.

Breakpoints

A breakpoint is a reference marker indicating a location where the execution flow is to be interrupted (a *break* in execution) and comes in both a machine-bound (hardware) variety and a program-bound (software) one, the latter being used as a substitute when the former is unavailable for some reason[2]. On a CPU-level, hardware support for breakpoints typically comes in the shape of one or a few debug registers where you specify the address at which to break and then have an ISR (Interrupt Service Routine) installed, attached to the debugger kernel. The software version is different in that the instruction at the targeted address is replaced with one that indicates a breakpoint[3], something that also excludes situations where the instruction resides on a memory page that is mapped as read-only. In regard to comparing hardware and software breakpoints, a rule of thumb is that the closer to the enabling machinery the breakpoint is defined, the more precise the results will be and most tools when faced with the decision will be biased towards hardware-supported breakpoints.

In principle, breakpoints are neither hard to grasp nor plan an implementation for, but the reality is that there are literally hundreds of details to take into account, some which lack robust solutions. Let's take a look at a few of the major ones to give some flavor to it all.

Concurrency – One kind of possibly concurrent execution scenarios come from the use of threads. These arrive in several forms based on level of abstraction, usage patterns, performance considerations and several other factors. There are different ways to implement threads, but a common denominator is that all threads share the same memory, as opposed to processes where each has its own[4]. When a debugger inserts a software breakpoint, it modifies the memory which, because the memory is shared, affects all threads. Since all threads can potentially execute the same code, which execution thread(s) should breakpoints effect: one, several or all of them? This kind of distinction requires better cooperation between the debugger and IDE, ultimately making breakpoints harder to define at build time (since the exact number of threads of execution and some means of identifying them, cannot reliably be determined at this point). Furthermore, even if we have ways of enumerating and identifying threads at runtime we will still have each thread trigger the breakpoint condition (unless the debugger takes the thread facility in account and triggers internal breakpoints at each context switch). Although we can return execution immediately if we trigger a breakpoint while executing an uninteresting thread, the chances for unwanted observer effects are considerably high.

Optimizations – The task of compiling code from a high-level language to a set of instructions close to a modern CPU architecture is a daunting one on its own

[2]Reasons for hardware supported breakpoints include insufficient privileges or registers already being allocated/in use.

[3]It is up to the debugger to make this modification and remember which instruction was replaced.

[4]Some real-time operating systems use the term *process* to describe threads.

accord. Add optimization techniques[5], and the end result barely resembles the original source code. Instructions have been reordered and scheduled. Expressions have been reconstructed and stripped from redundancies, reusing results from computation in amazing ways. Chances are that code present in the source view of the IDE no longer has comparable instruction at which to place a breakpoint. Complexity has increased by some magnitude without it even being deliberate.

Loading & Recompiling – Executing code is far from being some static entity; trampolines, dynamic recompilation and other runtime techniques all heavily contribute to changes on the executing code in unprecedented ways. Similar to optimizations, these kinds of functions increase the difficulty of connecting program descriptions such as source code to what is being executed and back again. Even with the possibility of injecting debug information to ease such a connection, chances are that some external library, third-party component or operating system feature will interfere in a way that is difficult to control.

In spite of these and similar problems, breakpoints continue to be a cornerstone feature from which most other high-level debugging functionality can be derived.

Step by Step

We can interrupt execution by exploiting several different mechanisms, for example software and hardware breakpoints. When execution flow has been successfully hijacked (or redirected), the case may be that you do not want to release this control immediately but rather plow through execution one instruction[6] at a time. Many algorithms for representing execution flow and data rely on this kind of functionality. With breakpoints in place, single stepping is just a specialized form implementable using the same technique but with additional complexity. When the initial breakpoint is triggered a new breakpoint is added at the next possible logical instruction. The increase in complexity comes from the meaning of *the next possible statement* which has to be deduced.

The fact that conditional branching and jumps make execution flow dynamic rather than a static computation from beginning to end means that the debugger must be able to analyze these kinds of instructions (and evaluate them based on state information) in order to determine where to break next. As many advanced debugger features require an evaluation engine of this kind anyway, in the grand scheme of things, it is not a major challenge.

In addition, as with most other features, there are several options when collaborating with hardware, either by toggling a single-stepping flag in the `CPU` or through hardware-based breakpoints, but `CPU` supported single-stepping is by far the most common. Other approaches are applied only in extremely rare cases.

[5] There are many of them.

[6] Or, at a higher level, this would be one statement or one line of code.

Watchpoints

While breakpoints as discussed are triggered by execution reaching a certain instruction during runtime, there is another side to the coin as well and that is data. A major task for programs is moving pieces of information around from memory to CPU registers and back to some other memory location again after being transformed as part of some calculation. It is typically not poorly chosen instructions by themselves that are harmful to a program but rather the state that they manipulate. Although CPU registers and flags are important containers for the state, they change at an unmanageable rate, and the value of the information they store is on a fine-grained scale diluted by leftovers from other execution necessities. Other kinds of storage, like memory, are by comparison an easier abstraction to deal with. This is where watchpoints, or data breakpoints, come in handy.

You specify an address and a size (usually ranging from 1 to 8 bytes, depending on underlying architecture) along with possible boundary conditions (read and/or write). When this condition is fulfilled, i.e., the address is being read or written, an interruption in execution will occur, similar to that of a normal breakpoint. This makes pinpointing a lot of memory-related problems easier, such as when you have data corruption from some wild pointer with unknown origin.

There are two comparatively easy ways of implementing watchpoint functionality. Preferred by far is to simply shift the problem to hardware. Some hardware architectures allow a small (1-4) number of watchpoints[7] to be set, but this may turn out too restrictive for some needs[8].

To implement watchpoints on a software-only basis, you expand upon single-stepping execution, copy the values at each watched address and then repeatedly compare at each consecutive step. This is a procedure with such an overhead that it becomes practically unfeasible for many targets where the ratio between capabilities and computational power of the debugger compared to the target (debuggee) is less than some magic value.

Conditionals

Navigating execution using a debugger is a time consuming, cumbersome and often repetitive task. It is easy to lose focus and track somewhere after the first few hundred instructions of single-stepping which calls for the use of automation.

From the described software implementation of watchpoints, we learned that it is possible (although at some cost) to attach a piece of code evaluating some

[7] These are part of / combined with hardware support for other kinds of breakpoints.

[8] This is especially the case as a source-level debugger may already allocate some of these for internal use.

expression and act only if the outcome of this expression yields true, such as when the value stored at a particular address is different from a preset one. In addition, reliably generating code at runtime for injection into a process is no longer considered a problem.

If we then combine breakpoints with some minimalistic user definable expression evaluation, we get *conditional breakpoints*[9].

With this in place we have covered the foundation of an *in situ* analysis tool, the debugger. We can *manipulate* the subject by temporarily replacing valid instructions with ones that act as controllable interrupts, or we can use hardware features for a similar effect, or a combination of both.

4.2.2 Representation

The debugger is restricted in the amount of control available to exert on a subject, debugee, and is therefore forced to cooperate with the surrounding environment (whether that is some `CPU` extension, `OS` function or `VM` hooks) in order to manipulate execution flow and gather data. That much is clear. A debugger is a user-centric tool, which implies that very little action is taken without direct instruction from the one doing the debugging. Therefore, quite a lot of consideration is taken considering *how* these instructions are given and interpreted and *how* results are presented.

It is time to look at the representation part of the debugger tool and what is involved when calling it *source-level*, as it is somewhat more complex than just hooking up to source code. Source-level debuggers built into IDEs (being somewhat of the norm compared to stand-alone versions) typically contain at least two levels of representations: assembly language and program source. However, more specialized ones targeting flow diagrams, domain specialization and similar things are far from uncommon.

Disassembly

Let's start at the lower levels and work our way upwards. Assembly language representation, at first glance, seems trivial; Processors execute instructions from addresses pointed out by a register called program counter (`PC`) or instruction pointer (`IP`). These instructions are binary encoded but separable into two distinct parts: the *opcode* and the *operand(s)*[10]. An opcode directs what is

[9] "Conditional breakpoints" is somewhat misguiding in that it suggests that it will only be activated when some particular condition holds true, but the condition actually follows the breakpoint, not the other way around. The breakpoint will be invoked every time but will only yield execution to a user interface when some condition is fulfilled

[10] In essence, there is also an even lower level on which to control some CPUs on for reprogramming or patching instruction sets and microcode, but the uses for this outside the world of an engineer specialized in processors can be considered minimal at most.

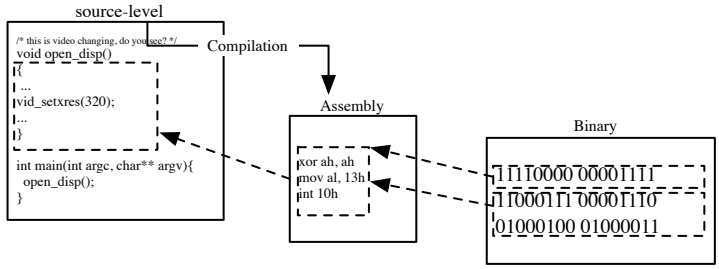

Figure 4.1: Assembly/disassembly

to be done (desired state transitions) while the operands regulate the details, which are similar in principle to instructions/functions with their respective parameters. The set of possible operations on some `CPU` is called quite plainly an instruction set.

Now, remembering binary patterns and combining them in various ways to perform larger and more complex calculations is quite cumbersome, as it is not really part of the computer science training curriculum; letters, digits, words and phrases are for some reason more intuitive than 1s and 0s for the vast majority. Therefore a common learning aid from other areas, *mnemonics*, is used; `11001101 11001100` on the `x86`, for instance, is far easier to remember as `INT 03`. A collection of mnemonics covering the span of a particular instruction set is grouped into an assembly language[11].

The first step for assembly-level representation in a debugger is therefore to map these binary patterns to their corresponding mnemonics and present the results to the user in an orderly manner. A two-stage decoder that in the first stage extracts opcode and runs it through a look up table with entries on opcode pattern and corresponding instruction name and in the second stage extracts operand patterns in a similar fashion would do the trick – if it wasn't for some complicating factors (which may or may not be present depending on the underlying `CPU` and memory architecture):

- Code/Data Conflict – The distinction between code and data when looking at what is being stored at a location in memory may be somewhat misguiding, simply because the von Neumann architecture allows for the same data store to be used as either code or data but also possibly both at the same time. Additionally, a program is allowed to reconfigure itself, which means that the code is self modifying.

[11] There are more additions brought on by assembly languages, but the focus now is on disassembly, representation and debugging.

- State sensitive Decoding – Some architectures have several different operating modes which change to some degree how instructions will be decoded and executed based on some runtime property. `ARM`, for instance, may have two or even three different modes (`ARM`, `THUMB` and `Jazelle-DBX`), whereas `ARM` and `THUMB` mode differ on, amongst other things, different fixed instruction length (32 versus 16 bit).
- Instruction length – The subject of the actual size of instructions has been purposely avoided up until now. In essence there are two different schools of thought on the matter, one being that instructions should be of a fixed width or size dimensioned to encompass all possible operands for all possible opcode and the other being that the size of an instruction should depend on the particular operand. Just considering disassembly and debugging, a variable instruction length introduces two additional complications: not being able to reliably detect if the contents at a particular address are part of an opcode or an operand and, by extension, not being able to reliably disassemble backwards[12].
- Undocumented instructions – Presumably a fading problem only prevalent in rare legacy cases, instruction sets seldom span the entire range of possible values covered by their size. This means that there are bit patterns that a processor should not be able to execute and instead when encountered trigger an *illegal instruction* trap or something to that effect. Some implementations however do decode and execute these undefined or undocumented instructions, but also do it in a repeatable manner with some controlled state transition being performed, making it fair game to take advantage of. The once popular 6502 processor has some notoriety in this regard among enthusiasts. In addition, and this applies to the state sensitive decoding problem as well, even though some bit patterns may not be covered by the instructions defined in a particular instruction set, they may still be allocated or reserved for use by some coprocessor.

As we are debugging, it is also somewhat safe to assume that when the need for disassembly appears (at a break-/watch- point, trap or exception), chances are that the PC points to a memory location where the contents are not valid code. Although that many cases are fairly easy for a human to decipher *(it is incomprehensible to me, therefore it must be some of that crazy computerspeak!),* the very point of these tools is to automate the tedious, the repetitive and the mundane. To reconnect back to the fundamental interventions of debugging, execution flow manipulation through breakpoints, recall that even representation on assembly-level has a fair disparity when compared to the real code in that whatever instruction used as a triggering mechanism for a software breakpoint may also be replaced by the original instruction in the assembly view, in effect obscuring them.

[12]This means starting at a known instruction and determine the instruction at some address lower than the current.

```
 1  int main(void)
 2  {
 3    char* buf = (char*) malloc(4096);
 4    int (*func_ptr)() = (int(*)()) buf;
 5    sprintf(buf, "life␣on␣the␣heap␣is␣grand\n");
 6
 7    return func_ptr();
 8  }
 9
10  /* sample disassembly at crash-point (gdb> disassemble startaddr
        endaddr)*/
11  insb   (%dx),%es:(%edi)
12  imul   $0x206e6f20,0x65(%esi),%esp
13  je     0x800072
14  and    %ch,%gs:0x65(%eax)
15
16  /* sample dump of contents at crash-point (gdb> x/s startaddr)*/
17  life on the heap is grand
```

Figure 4.2: Two different representations (lines 11..14 versus line 17) of the same memory region at a crash

The Source Connection

While assembly view may be needed in order to look at the fine-grained details of some smaller block of code for whatever reason, it is still comparatively somewhat of a corner case when dealing with software written mostly in some high-level language. A more intuitive view is probably that of source code. Looking again at the Figure 4.1, if we block out the source code when looking from binary to assembly, how can we make the last reverse connection back to source code? With only the binary encoded data available, it is impossible.

Looking at source code and looking at linking, we are literally covered in a variety of symbols. When staring at a straight disassembly of the produced binary, however, the question of where all the symbols have gone quickly springs into mind. The answer to this question is that they were lost in compilation. While being a great aid for a developer, once resolved the actual symbol names do little more than waste space from a computer point of view and are therefore often discarded unless the tools involved are instructed otherwise. Specifics on symbols can be tucked away in the binary itself or in a separate database, and there is a variety of formats suggesting how this can be done in order to optimize some parameter like storage space or lookup time, both of which are pretty essential due to the sheer amount of symbols and the size of accompanying data.

A simple symbol table would contain at the very least symbol name, the address at which the data resides and what type the data represents. The specifics to each also apply (such as padding, alignment, size, subtype, etc.), but some

meta-level information is also needed, e.g., reference to source code file and the line number at which the symbol definition can be found, both of which are necessary in order for a debugger to close the loop and successfully map low-level information back up to source code. These kinds of details become quite gory for languages with a complex symbol environment like `C++`. To cope with these kinds of problems and to introduce portability between compilers, languages and environments, there are quite a few standards available, all of which are somewhat outside the scope of this section. For an exercise to illustrate the extent of debug formats, correlate [DWARF] to the `ELF / PE` binary formats.

The algorithms and structures that constitute a compiler can be weighted and configured to fit a variety of purposes. When the purpose behind a particular configuration is to make the information a debugger can present more reliable, the outcome is rather different than it would be had the purpose been to make the code run as efficiently as possible in terms of space and time for a particular hardware set up. These two very different ends are for most situations in a state of conflict. A binary produced to fit the former purpose is colloquially referred to as a debug build.

Call It a Stack

A debugger can show which instructions lie at, in front of, and possibly also before a particular breakpoint. It can also provide information on the register state and about whatever is being stored in certain memory locations when such a breakpoint has been triggered. If the locations in question are covered by data extracted from a debug information format, the connection can also be expanded to specific lines in source code at the involved declarations and statements – if the information provided is accurate, that is. All of these are various levels to show where we are in execution at that very moment. This may be helpful on its own, but chances are that we need not only to study data from a specific point of view or *context* but also examine some execution history to answer the questions *where did we come from* and *how did we reach this point?*.

The *imperative programming paradigm* is, as implied, fairly straightforward; it is merely a series of *do this then do that* and is as such low-level programming in its easiest of forms covered by assembly languages. In machine-level programming, there is quite a few simple instruction patterns which helps to cut down the amount of strenuous and repetitive programming that has to be done. Loops, procedures and functions in higher-level imperative programming languages are mostly abstractions made from the most frequently employed instruction patterns. The way functions are implemented and defined typically allows for both chaining several functions together (calling a different function from within a function) and for defining a function using its own

definition from within itself (somewhat harder to wrap your mind around, but this is a programmer's take on recursion).

The code generated from compilers in order to implement language-specific interpretations of the idea of functions – looking at the case of a C compiler on an `x86` processor that first and foremost uses the stack facility of the processor meaning the two registers `ESP` (stack pointer), `EBP` (frame or base pointer) and the instructions that can manipulate these (push, pop, etc.). `ESP` points to the last stored element on the stack(a `PUSH` instruction, for instance, would first have to decrease the value of `ESP` then write the data), while `EBP` points to the current frame within the stack.

The idea and distinction between `EBP` and `ESP` is that `EBP` would be a known reference point only explicitly modified while `ESP` is changed implicitly by several instructions and is therefore harder to keep track of. Invoking a function can be subdivided into three steps: setting up the arguments, handing over control over execution to the function and finally restoring the control of execution. Restoring execution control also involves passing function return values back to the calling code.

There are several steps to performing a function call and it can be done in different ways all with their respective merit. Some are shrink-wrapped into a package called simply a *calling convention*, and many languages support several such conventions. Let's look at the default one of the C programming language:

prologue

1. Push argument on the stack, right to left.
2. Call the function (`x86 CALL` instruction stores instruction pointer/`PC` address of the instruction after the call instruction and then jumps to the address of the function).
3. Save current frame pointer to stack (`PUSH EBP`), and set to the same value as the stack pointer (`MOV EBP, ESP`).
4. Save any registers that will be used to stack (`PUSH reg` repeatedly or `PUSHA`).
5. Reserve stack space for local variables by modifying `ESP` (`SUB ESP`, *space to reserve*).

After a completed function prologue, the current stack layout will look something like Figure 4.3.

epilogue

1. Restore saved registers. `EAX` (and in some cases `EDX`) is used for storing the function return value.
2. Restore base pointer from stack.

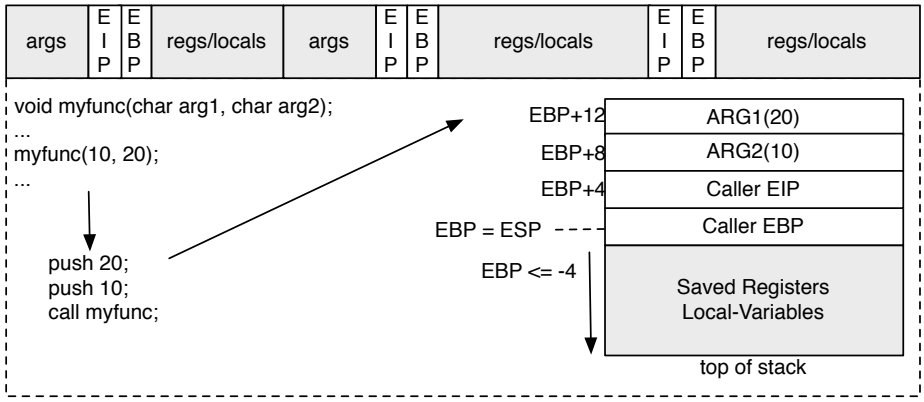

Figure 4.3: Stack layout after prologue

3. Return (`x86 RET` instruction pops `PC` from stack and returns to this address).

4. Clean up old arguments from stack (responsibility of the calling function).

From this follows that if we are at a breakpoint placed within a function, we can retrace stack conditions throughout the entire chain of calls by using all the frame pointers of previous functions spread throughout the stack as points of reference; this is also called *unwinding the stack*. Combined with debug information it is possible to show which functions were called and with what arguments, but also to switch context to a specific function along this chain, examining the state of local variables at the point of the call. This is a very powerful and detailed information source[13].

As with other parts described, there are complications which makes the actual implementation of this idea a lot more troublesome than what it appears to be at at first glance. As with disassembly, chances are that we will have to try and make the best out of a situation where the system is already in a state of disarray and not far from total breakdown. That also means that parts of memory cannot be assumed to contain what it is supposed to. This includes stack contents, which implies frame pointer contents as well.

Let's look at an example of memory corruption on the stack:

[13]It may be worth noting that stack unwinding is not a technique used exclusively for debugging; it may be needed as a way to deal with some language specifics like C++ exceptions, C longjmp and signal handling but in a more simplistic form.

```
1  void myfunc(char* arg)
2  {
3    char dptr[32];
4    memset(dptr, 0, sizeof(dptr));
5    strncat(dptr, arg, sizeof(dptr));
6  }
```

Figure 4.4: Strncat off-by-one

This is an obvious one. The size argument to the `strncat` function concerns the number of characters to copy, and the trailing `NULL` character is not counted. For the case where the input string is equal to or larger than the stack-allocated buffer, an extra byte will be written beyond the allocated range for `dptr` on the stack. Because the stack grows downwards and write operations like in the example start at a base address and works its way upwards, this extra byte will be written to the prologue- important data. In this case with no other registers saved on the stack, it will be the `EBP`.

The overwrite is just a single byte and `EBP` occupies four of them (32-bit pointer), meaning that parts will be intact and only the eight least significant bits will be changed -in this particular case, zeroed out, most likely leading to a crash further down the line. Placing a breakpoint at the return statement on line 6 would then yield a broken stack trace unless the stack unwinder is a bit crafty. Even though a broken `EBP` in the stack is a telltale sign of something that needs to be dealt with immediately, the reason for the corruption is probably far more complicated than in the example. That is why considerable effort is placed on trying to stack unwind, even though the stack may be broken. Trying implies that we cannot guarantee success, and there is a fair amount of system specifics that may either improve or worsen conditions for the debugger. In short, the techniques involved in this regard are a mix of instruction simulation, heuristics and forward chaining using dependency graphs.

The single most contributing factor to the difficulty of stack unwinding is not the effect of a particular bug in a limited context, but rather diversity of optimizations. One thing about conventions is that they are just that: polite suggestions that something ought to be done in a certain way and if a particular situation deviates from some agreed-upon convention, there might be consequences to pay, but the convention itself is per se not enforced. This leaves quite a bit of wiggle room for compilers that have access to all these extra details on how some functions are used internally, and if it can be determined in advance that a set of functions will not explicitly need the frame pointer for anything other than following the calling convention, that part may simply be stripped.

Another problem of identifying convention in order to retrace steps is that there are often many similar, but not identical, ways of achieving roughly the same state transition. Standard C calling convention and others define information

that need to be on the stack and in a specific order, not how the information is supposed to get there in the first place.

The reason that much effort is put into making a debugger able to take situations like stack unwinding with optimized code and others into consideration, is the same reason behind using a debugger in the first place: a subject is behaving in an unexpected way and therefore cannot be expected to follow convention. Any and all information that can be extracted from the broken system may be key in solving the underlying problem, but it is also of utmost importance that this information is truthful. It may be both difficult and time consuming to externally verify the truthfulness of debugger output, when at all possible.

4.2.3 Interfaces

Mentioned time and again, the debugger needs support from the immediate environment in order to exert control over a subject. The high-level prerequisites from the initial description of a debugger stated that it has to be able to control target execution and sample, possibly change parts of the target state, decode the meaning of the sampled state and do all this at the appropriate point in time.

```
1  int main(int argc, char** argv)
2  {
3    int pid = fork();
4    int waitfl;
5
6    switch (pid)
7      case 0:
8        ptrace(0, 0, 0, 0);
9        execl(argv[1], "debug_child", NULL);
10     break;
11
12   default:
13     wait(&waitfl);
14     do
15     {
16       ptrace(PTRACE_SINGLESTEP, pid, 0, 0);
17       wait(&waitfl);
18     } while(!WIFEXITED(waitfl));
19
20
21
22   return 0;
23 }
```

Figure 4.5: Barebones example on ptrace usage

Hardware Interfaces

Proper `ptrace` support does a great job at providing the amount of control needed in order to implement a debugger targeting userland, but does leave a fair amount of tracks and causes enough of a stir to possibly trigger several complicated observer effects. In the end, the control that a kernel provides may simply be insufficient or even unwise to use in the first place. The alternative then goes to the next environment on the list – the enabling machine. While we are already in practice working with the machine partly due to hardware support from hardware-enabled breakpoints, there is a different way to go about that. Although somewhat costly, it may prove a lot more efficient at its job.

In many other areas with some sort of diagnostic component as part of the daily work routine, as is the case for physicians and mechanics, the idea of having some other device foreign to the target is rather the norm. The way we can go about it with software debugging software that is executing within the confines of the same machinery is comparatively speaking something rather odd, and perhaps we are making the job more difficult than necessary.

Now, which possibilities does running a debugger on its separate hardware have to offer[14]?

A pretty obvious benefit at the hardware-level is that you may have access to data that could otherwise be lost. While a crashing OS kernel might be able to generate a core dump for many problems, driver development and edgy, unstable hardware subcomponents may not provide the window of opportunity needed to generate and store a dump before execution comes to a halt.

Another one, perhaps is a bit more subtle, is the reduced dependency on shared resources; a debugger executing on the same machinery as the software it is supposed to manipulate, is forced to share at the very least both memory and `CPU` with its subject, thus any bug that is in part or in its entirety, dependent on some particular `CPU` and memory, becomes quite elusive.

In a more complicated software environment consisting of multiple concurrent resources, hundreds of mostly independent processes such as typical desktop software where several less obvious resources are shared. Consider for instance that a `ptrace` based debugger has an implied context switch each and every time the debugger is invoked; these are comparatively costly and intrusive. In addition, having a conditional breakpoint placed in a busy part of a program will add considerable overhead.

In this respect a debugger based on separated hardware is a lot more transparent but it is dependent on some golden ratio between source/hardware capacity and target/hardware capacity. When this ratio exceeds some value to

[14] This does not directly imply a case advocating for the use of remote software debugging as that particular case still shares the same general problems of doing it from within the target system, with some new ones added for good measure

the advantage of the hardware debugger, new and interesting possibilities are opened up. The main one is the ability to completely track all state changes within the context of the machine for some limited period of time, in essence creating millions of miniature snapshots allowing you to rollback to the state of a previously executed state at a granularity of individual instructions, effectively allowing execution to be reversed. This is particularly feasible and interesting for embedded developers where the computers used for development often overpower the target device by several magnitudes in terms of both computational power and storage space.

When it comes to interfaces, how do hardware debuggers connect to a target? As stated earlier, a debugger needs support from the immediate environment in order to exert control over a subject. Displacing debugging down one level of abstraction does not remove this prerequisite: the only thing that changes is whatever that comprises the immediate environment. Environment support at this level is the debugging capabilities of the CPU and other components in the target hardware when exposed as a hardware interface. Another environment even further down the line would be that of communication buses. One hardware interface that exposes the debugging facilities of a microprocessor is described by the IEEE/1149.1 standard, also known as J-Tag, which dates back to 1985 and was primarily designed to facilitate testing.

Due in part to age and lack of detailed specifics on implementation, building a debugger using J-Tag without vendor-specific, presumably proprietary, information is a futile endeavor for any purposes beyond the trivial. Work on standards addressing this problem is underway, a notable such effort would be that of Open-Core Protocol[OCPIP08], but wide-spread adoption is likely to take considerable time. In spite of its potential, there are some considerations for external hardware supporting debugging that reduces chances for success.

- Cost – Dedicated hardware capable of attaching to other platforms is a large development endeavor in itself, targeting a comparatively small consumer base. This makes dedicated debugging hardware costly, if available, and may not be entirely compatible with the specific demands placed by a custom developed embedded system.
- Inherent Limitations – As hardware debuggers are embedded systems in their own right, there are some limits to capacity in terms of computational power but also in terms of memory and buffers for storing measurements and intermediate target states.
- Protection – Popular embedded systems like cellphones, video game consoles and others are subject to both piracy and external influences attempting to alter the system for some undesired purpose. It is therefore a common thing to embed various kinds of safeguards against such intervention and an easy such safeguard is simply to not have any accessible interface for attaching hardware debuggers.

4.2.4 Discussion

Concluding and summarizing the section on debuggers, let's iterate on the fundamental problems faced when trying to use some flavor of source-level debugger:

- Code-reference – With source code as the final representation for measurements, it is of the utmost importance that the connection between low-level states and the source-level view is as representative and accurate as possible. This can be hard to achieve when the build process for constructing the executing system involves additional layers between source code and the final executable, but also when the programming language has facilities that expand into considerably different dynamic constructs, such as advanced forms of `C++` templates.
- Optimization – A primary task of compilation is optimization, which is expected to become even more involved as target capacity for parallel execution expands. Any future development in terms of optimization also widens the gap between builds targetting debugging and builds targetting deployment.
- Overhead – The kind of intervention and measurement that debuggers perform comes at a high overhead. Thus, considerable probe effects and the means for reducing such overhead have not improved notably from the initial conception. On the contrary, the sensitivity of execution performance has increased with modern optimization techniques both on the `CPU` but also on the compiler-generated code.
- Anti- debugging – As is the case with hardware debuggers, protection techniques against attaching a debugger to the target are frequently employed. Software- implemented techniques for preventing a debugging from attaching and adding breakpoints are numerous and bothersome, ranging from mere obfuscation to exploiting the same mechanisms a debugger would use and override, to implementing intended functions of the software, such as having system execution relying on the `ptrace` interface being active or running entirely within exception handlers.

4.3 Tracer

We have already looked at a very specific case of tracing as part of a debugger in the shape of stack unwinding in order to generate a call trace, but it is comparatively rather an exception than the norm; getting a trace of function call in an in-depth manner by unwinding the stack (with the overall goal of decoding and representing data stored on the stack) is somewhat of a backward chaining[15] activity. Tracing in a variety of forms is already part of many

[15]Reasoning from a conclusion backwards to reaching a premise.

debuggers, not only as they are slowly turning into some all-purpose Swiss army knife component within an IDE, but also because mechanisms used for performing a trace are already implemented and therefore present in a debugger. The foundation of tracing when looking at it from a more abstract level is, however, interesting enough to warrant further dissection. It is also, a practice employed often by most programmers.

The main features of a source-level debugger such as breakpoints and state space exploration can be considered focusing on the specifics: getting a highly detailed view of a smaller subsection of a system at a highly detailed level of precision. Tracing, by comparison, is from a different school of thought in that you employ the idea of looking at something more generic, such as the path execution takes throughout the course of a function by taking note of branching made in effect of conditionals in a recursive fashion, revealing dependencies between functions and highlighting code paths that rarely get triggered. The study of functions or explicit code paths is not necessary. However, any connection between some part of a software system and the flow of execution may be of interest.

The mechanisms of debugger-aided tracing are similar to that of breakpoints placed at branching-points and at function prologues/epilogues, but instead of pausing execution flow awaiting user input a log entry of sorts is generated, containing at least some kind of identifier and possibly a sample of relevant state. The dependency on some predetermined information format mapping code and memory addresses is also similar to that of regular debugger breakpoints. Representations of the data gathered would be anything from a simple sequential list of the identifiers of log entries to something more advanced like a graph showing branches and interdependencies.

The most popular form of trace however is far more simple in construction: inserting a few output statements that writes some strings into an output device or file. While this kind of debug output may have other uses as well, such as log-messages for some intended end user, its prominent use during development, especially debug builds, is the interesting but often overused trace-like behavior.

While just looking at data from some particular context, the basic functionality provided by a source-level debugger, is of good use in situations where the points of interests have been narrowed down to a few specific suspects. Tracing may have some merit even at this phase because of its particular effect to concurrency which warrants some additional explanation:

Having several threads executing either same or similar code using a shared dataset opens up a world of trouble. If done within the same context of execution on an overarching level, like the one provided by the operating system processes, any and all protection for various situations of conflict that may arise are left to whatever thread management facility a programming language runtime may provide – along with safeguards explicitly added by the programmer.

In spite of concurrency concerns, some programmers have an inclination towards the extensive (ab)use of threads with motivation akin to threads being a more intuitive or natural way of achieving parallelism in software systems especially when dealing with user interface or asynchronous I/O programming.

Debugger and by extension IDE support for multiple threads of execution within a single program is somewhat harder to implement, but particularly lacking in representation. While some thread manipulation is exposed through a debugger user interface for known thread management facilities, it revolves around support for running/suspending/breakpointing and re-prioritizing specific threads, not looking at how they interact within a particular region of code. In this respect tracing can address these representational shortcomings somewhat and both assist and thwart analysis efforts in the process.

```
1  #include <pthread.h>
2
3  void* outp(void* arg){
4    for (;;)
5      printf("%s\n", (char*) arg);
6
7    return NULL;
8  }
9
10 int main(){
11   pthread_t t1;
12
13   pthread_create(&t1, NULL, outp, (void*)"1");
14   outp("2");
15
16   return 0;
17 }
```

Figure 4.6: Two concurrent threads of execution running within the same region of the code tracing state to a shared output

The code in Figure 4.6 would yield some output alternating between 1s and 2s in some order that may change between each consecutive run depending on thread scheduler behavior and possibly on other sources[16]. Whatever underlying technique for tracing is used, having some kind of identifier denoting a particular thread of execution in order to separate respective output in the resulting data flow, would be necessary.

[16]Had the outcome from concurrent execution been solely dependent on thread scheduling, the problem presented would be of a more academic nature.

4.3.1 Discussion

Like other tools for extracting information about software and software execution, a tracer is impaired by a series of shortcomings inherent in the underlying process.

The main benefits of tracing are paradoxically enough also a possible disadvantage. Overhead depends on the amount of tracepoints inserted, how often they are triggered, the amount of information sampled and exported from the system, and whatever limits are imposed by actually exporting. These dependencies are not surprisingly much different from those of conditional breakpoints with the main difference being representation. Increasing the amount of tracepoints will increase the precision and amount of data, therefore also increasing sampling overhead and the likelihood of observer effects. An exact trace is virtually no different than a whole system snapshot.

Additionally, whatever the facility exploited to export data may be, if this facility shares mechanisms with the cause of the effect in question, the likelihood of observer effects increases yet again while the precision and reliability of the trace information decreases. This does not exclude tracing from the list of available approaches, however it merely highlights the importance of choosing a good mechanism for exporting and sampling data.

Any and all debugging techniques come with a certain additional risk when dealing with an uncooperative subject where anti-debugging techniques are employed. These, however, are not generally systemic but rather are directed to some specific way of implementing breakpoints or some other fundamental part of the debugger. Whatever conditions apply, chances are that you may need to ascertain the validity of the context available at a triggered breakpoint and correlate it to a trace generated using a different mechanism.

In closing, the principles of tracing can be used for analyzing aspects that are troublesome at best when employing detailed controlled dissection of the program state, especially when looking at performance problems and profiling.

4.4 Profiler

Performance analysis, or *profiling* in software jargon, is the final part of the debugger tool suite that will be covered herein, and the only one with fundamentals grounded in statistics. While the other parts concern either the study of detailed internal states at a specific point in execution or exploring sequential patterns, profiling revolves around measuring the usage of some specific resource and correlate to either the system as a whole or some subdivision over a period of time.

From this description we can derive some more detail about the necessities of profiling: targetted resource, entry/exit points and time stamp. None of these

are particularly surprising. The targetted resource concerns which of the few available shared resources in a computer that is of interest to the particular profiling session. CPU, disk, memory are the main three distinct resource categories, however the targeted resource might just as well be on a more abstract level, such as the usage of some internal data structure. Time stamping may be in whatever unit of measurement feasible, but is typically regulated by some external tick. Entry-/exit- points depends entirely on what is to be measured. The idea is that they should encompass the scope of the recurrent event to profile, pruning anything that may distort or reduce the precision of the result.

Most operating systems provide a profiling facility on a per process level of granularity as part of process control using input from the process scheduler, allowing a user to determine if a suspicious process may be caught in a livelock or similar problem as basis for terminating rampant tasks.

In essence, though, any source that might be used as a tracepoint could also be a feasible entry or exit point for profiling. Going about actually gathering the samples, there are two somewhat distinct ways of approaching the problem; event, statistical or frequency driven sampling.

- Event driven sampling – This corresponds to what was just mentioned about using tracepoints for profiling. Combining a trace-sample with a time stamp is sufficient for profiling assuming that tracepoint used as entry has a semantically corresponding exit tracepoint with which to form a pair.

  In automated profiling as part of regular debugger-provided profiling, the same information used to map the address of a breakpoint to some corresponding line of code within a function can be used to form a pair of tracepoints from function prologue and function epilogue, respectively, hopefully providing better results than OS scheduling may.

  In addition to the dependency on details required regarding the specific software, event driven sampling in this regard has a notable overhead, proportional to the occurrence of phenomena to measure which, combined with the problem stated in the section on tracing, means that we are relying on and affecting the very target of our measurements.

- Frequency driven sampling – A bit more situational than event driven sampling, frequency driven sampling uses considerably less system-specific details and can be implemented and run on builds lacking other debug data. This is particularly useful at a hardware level and can have a performance impact several magnitudes less than regular event driven sampling.

  While technically still an event, but one triggered independently of the target to measure, frequency driven sampling is driven at a regular rate determined by some clock pulse. At each tick some state holder with known contents and location is sampled, such as the instruction pointer of a CPU.

4.4.1 Discussion

It is fairly complicated to get a good grasp on software performance. Trying to draw conclusions from any measurements taken on some part assumes that the data extracted is both generalizable and commensurable, two heavily loaded topics of philosophy.

The data needs to be generalizable for it to represent the performance of the system as a whole. It needs to be commensurable because the results gained from one run of the system have to compare not only against results from repeated runs and results from different instances with slight variations in environment, but also between different versions of the software code in order to show that some particular performance problem has been dealt with. Generally speaking, if there's one thing that varies to an exceptional degree between different execution runs, it is performance.

This is in itself not particularly surprising; lots of modern features added to compilers, language runtime support and operating systems are attempts to improve execution performance and features such as dynamic recompilation (JIT), garbage collection, on-demand linking and resource scheduling, all contribute to making execution performance even more difficult to ascertain.

References

[GDBINTERNAL] Free-Software Foundation / GNU Project, GDB Internals, available from http://www.gnu.org/software/gdb/documentation/

[ROSENBERG96] Johnathan Rosenberg, How Debuggers Work: Algorithms, Data Structures, and Architecture, JOHN WILEY & SONS 1996, ISBN 0471149667

[DWARF] DWARF Debugging Standard v3, available from http://www.dwarfstd.org

[CCONV] Calling Conventions – for different C++ compilers and operating systems, available from http://www.agner.org/optimize/calling_conventions.pdf

[OCPIP08] OCP International Partnership, OCP-IP at http://www.ocpip.org