Analysis of requirements for an automated testing and grading assistance system

by

Jonas Lindgren

LIU-IDA/LITH-EX-A--13/048--SE

2013-10-05
Master’s Thesis

Analysis of requirements for an automated testing and grading assistance system

by

Jonas Lindgren

LIU-IDA/LITH-EX-A--13/048--SE

2013-10-05

Supervisor: Klas Arvidsson
Examiner: Mariam Kamkar
Abstract

This thesis analyzes the configuration and security requirements of an automated assignment testing system. The requirements for a flexible yet powerful configuration format is discussed in depth, and an appropriate configuration format is chosen. Additionally, the overall security requirements of this system is discussed, analyzing the different alternatives available to fulfill the requirements.
# Contents

1 Introduction ........................................... 4  
   1.1 Abbreviations and terms ................................. 4  
   1.2 Background ........................................... 5  
   1.3 Expectations on reader’s knowledge ..................... 7  
   1.4 Purpose .............................................. 7  
   1.5 Method .............................................. 7  

2 Principles ............................................. 9  
   2.1 Flexibility ......................................... 9  
   2.2 Robustness ......................................... 9  
      2.2.1 Modularization ................................. 10  
      2.2.2 Possible risks ................................. 10  
   2.3 Trust .............................................. 12  
      2.3.1 Possible risks ................................. 12  

3 Security alternatives .................................. 16  
   3.1 Implementations in existing systems .................... 16  
      3.1.1 QUIMERA ........................................ 16  
      3.1.2 DOMjudge ....................................... 16  
      3.1.3 Online Judge .................................... 17  
   3.2 Implementation alternatives ............................ 17  
      3.2.1 Solutions not considered ....................... 17  
      3.2.2 No protection ................................... 18  
      3.2.3 Static analysis of code ......................... 18  
      3.2.4 Using a custom chroot solution ................. 19  
      3.2.5 System call interpositioning ................... 20  
      3.2.6 Using a virtual machine ....................... 21  
   3.3 Conclusion ......................................... 23  

4 Configuration format .................................. 24  
   4.1 Formats in existing systems ......................... 24  
      4.1.1 QUIMERA ........................................ 24  
      4.1.2 DOMjudge ....................................... 24  
      4.1.3 Online Judge .................................... 25
Chapter 1

Introduction

1.1 Abbreviations and terms

A number of abbreviations and terms will be used throughout this thesis, the meaning of which may not be obvious.

LiU Linköping University, a Swedish university located in Linköping.

IDA Department of Computer and Information Science, a department at LiU.

AES Authentic Examination System, a system used at IDA to administer exams in conditions close to real working conditions.

AESEA Authentic Examination System Examination Assistance, the proposed system being discussed here.

Sandbox A virtual container used to restrict access for and to separate running programs.

Walltime The real amount of time that a process has been running.

CPU-time The amount of time a process has spent actively running, as opposed to sleeping or waiting for a blocked resource.

Inode A data structure used in filesystems to store information about a file object.

Process An instance of a computer program that is being executed. May be composed of multiple threads.

API Application Programming Interface, the specification for how to interact with, for example, a library.

Chroot A UNIX-command for changing the apparent root directory for the current running process and any children. Processes altered this way cannot normally access files outside the new directory structure that is established.
Kernel The kernel is a fundamental part of the operating system, managing processes and handling system calls.

Virtual machine An abstraction of underlying hardware implemented in software. Allows a program or an entire operating system to run on emulated hardware, as opposed to the actual host hardware.

Exam For the purposes of this paper, an exam consists of a number of students solving programming tasks on computers under controlled circumstances for a limited time. The students can submit their solutions via a computer application, and receive immediate feedback as assistants in another room grade the submitted solutions and respond via the same computer application.

AJAX Asynchronous Javascript and XML, a combination of techniques commonly used in web development.

.NET CLR The .NET Common Language Runtime is a core component in Microsoft’s .NET, handling Just-In-Time compilation and execution of programs.

Superuser A special account for system administration, the superuser account normally has full, unrestricted access to all parts of an operating system.

1.2 Background

At LiU IDA is responsible for several large programming courses, during the examination of which students are given limited programming assignments, each to be solved as a standalone program. These examinations are often graded live using the Authentic Examination System, which allows the students to write their solutions, compile them and to submit the solutions to be graded while the exam is still ongoing. The grading is taken care of by assistants, who may then report the results back to the student in question, letting them know that they passed the assignment or that they, if the rules permit, may have to try that assignment again. This system can also be used to facilitate communication between the course examiner and the students.

Figure 1.1: Current AES workflow
These assignments are often similar, yet diverse in how they need to be approached for a solution and the input the require and the output they provide.

The grading of each assignment requires a careful investigation of both the functionality and the structure of the program. The assistant has to make sure it performs as required, with no bugs in unexpected circumstances. The assistant also has to ensure that the code fulfills any style and best-practice requirements placed upon the students. As this is done while the exam is still ongoing, with a potentially large number of students waiting to have their assignments graded in a timely manner, this can quickly become a stressful and repetitive task.

It would be useful to have a tool to aid in grading these assignments by adding an extra test-step before each assignment is manually graded. This test-step could handle many of the rote testing tasks that an assistant would otherwise have to perform, as well as provide more exhaustive testing in a shorter amount of time than any assistant could possibly manage. Such a system could also be used in a lab setting, where students might be asked to pass an automated test prior to demonstrating their work to the lab assistant. This tool, to be used in conjunction with AES, would provide support for the assistants to allow them to provide faster and more accurate grading of assignments.

![Proposed AES + AESEA workflow](image)

Figure 1.2: Proposed AES + AESEA workflow

While there are plenty of similar systems such as programming competition judging systems or lab grading aid systems already in use, for example KATTIS[6] already used here at LiU, none of them quite fit the use-case of the proposed system, and would require extensive modification to do so. As such I am examining the viability of a proposed new system, the Authentic Examination System Examination Assistance, with this thesis as a major first step in that evaluation.
CHAPTER 1. INTRODUCTION

1.3 Expectations on reader’s knowledge

This thesis is written for a reader possessing the understanding of at least a first-year Master student in Computer Science. Familiarity with programming languages and operating systems is expected.

1.4 Purpose

The purpose of this thesis is to examine what features the system proposed in the background would require, as well as what considerations have to be taken into account when designing this system. The focus will lie on extensibility and reliability, as a system that cannot handle all of the situations it is placed in will not be of much use, and a system that breaks down, or worse, can be subverted by a user, cannot be trusted to be used when real grades are on the line.

Toward that end, I will seek to answer the following questions:

1. What principles should this system be designed after, and how do these influence the other choices?
2. What are the security considerations this system needs to be designed to cover?
3. What are the available means to handle those security considerations?
4. What configuration format provides the greatest ease of use and flexibility?

These questions will be answered within the limitations inherent to the environment of LiU:

1. The system must work within LiU’s computer systems without major alterations, operating systems or otherwise, of those systems.
2. The system must not be reliant upon any service or license external to LiU.
3. The system must be maintainable and expandable by LiU personnel.

1.5 Method

The conclusions of this thesis will be based in part upon theoretical studies and in part upon practical studies. First I will investigate what principles I need to apply and what these tell us about the security implications. I will then proceed to examine these security risks further, highlighting possible ways to manage these risks and then examining these ways further.

Other, similar systems will then be compared with each other to highlight commonalities, flaws and wise choices.

Finally I will investigate the possibilities for configuration formats for AE-SEA, beginning with a broad comparison of several candidates. Once the field
is narrowed down, a theoretical and practical, in-depth comparison between the candidates will take place, followed by a conclusion.
Chapter 2

Principles

In order for the system to be considered useful and reliable in its role, beyond the basic functionality of a the system, three fundamental requirements that have to be fulfilled. I will investigate each separately.

2.1 Flexibility

Not all assignments are the same, and for the system to be useful across as broad a range of assignment as possible for as many programming languages as possible, the system has to be designed with flexibility in mind.

The programming languages that AESEA needs to be able to work with includes:

- Ada A statically typed, imperative programming language.
- Java An object-oriented, general purpose programming language.
- C/C++ A general purpose programming language and its object-oriented relative.
- Python A general purpose, high-level programming language.
- Lisp A high-level programming language originally based on mathematical notation.

In addition, the system also needs to be able to deal with the various types of assignments for grading, found in the use case documents for AESEA[29].

2.2 Robustness

Since the system will be used to grade assignments the students wait during an exam, the grading is fundamentally a time-critical task. To ensure there are no unexpected interruptions or downtimes it is therefore crucial that the system is robust, and will not crash or otherwise stop working while an exam is ongoing.
How to minimize the amount of interdependencies that can cause a failure in one place to cascade into an overall failure of the system, and how to design each individual component of the system in such a way that they are all individually robust?

Additionally, a system that requires expert knowledge or large amounts of effort to maintain and use has a strong tendency to be displaced and forgotten. A robust system is therefore easy to use, maintain and understand, to ensure that it remains in use for as long as it is useful.

2.2.1 Modularization

Whenever robustness of a complex system is prioritized, breaking the larger system up into a number of smaller modules is desirable. This is a design pattern that has been implemented in the past with great success for operating systems such as Linux and other, related operating systems[38]. Modularization of AESEA into a number of smaller, independent modules that can be combined via simple shell-commands into a larger whole will serve to minimize interdependencies, while at the same time forcing the modules to interact via strictly defined interfaces. This will serve to keep AESEA stable, even if an individual module contains bugs or becomes unusable due to changes in the program environment, the configuration of the system can be adjusted to work around that particular module. This assures the continued functioning of AESEA overall, sans the functionality of the malfunctioning module of course, until the malfunctioning module can be repaired or replaced. Likewise, if a bug in a module causes a crash or offers an opportunity for an exploit, this still only affects that particular module and won’t necessarily subvert or bring the whole system down. An overview of the intended modules of AESEA can be found in “AESEA Use Cases”[29].

2.2.2 Possible risks

To be classified as robust the system has to deal with not only the ordinary such as failed compiles and crashing programs, but also the extraordinary such as intentionally malicious programs. It is also possible that a legitimate solution may suffer from some extreme kind of behavior under certain conditions. The situations classified as risks toward the robustness of the system are listed below.

Denial of service: Forcing a high compilation time

Risk It is surprisingly easy to force a high compile time in certain languages, for example C or C++. Even a legitimate program may be designed in such a way that it takes a very long time to compile, and the difference between a correct, quick-to-compile program and one that will never stop compiling may be as little as one line of code.
Prevention  Something as simple as a compilation time limit would eliminate this issue. A good complement to this solution is to keep submissions in a processing queue while they wait for processing, and limiting how many submissions one individual user may have in that queue. One might also limit how many submissions from a specific user may be processed sequentially before another user is prioritized, to ensure all users get their “fair share”[2, Pg. 91-92].

Denial of service: Long execution times

Risk  Just as it is easy to force a high compile time in certain languages, it is always easy to create a program that will not terminate normally. Such a program can keep running, consuming as much CPU as it is given access to until forcibly killed, if ever. No malicious intent is necessary, a misguided novice programmer could easily forget a detail that causes an infinite loop under certain circumstances.

Prevention  Prevention of this attack is very similar to prevention of the attack described in 2.2.2. Something as simple as a CPU- or wall-time limit, coupled with a “fair-queue”-scheme of some sort would solve this problem.

Denial of service: Consuming resources at compilation time

Risk  Certain programming languages, for example C++, require lookahead when parsing code during the compilation of source code. By providing a very large file as part of the source code, the compiler may attempt to read the whole thing into memory, causing runaway consumption of system memory. This is especially troublesome when considering that forcing the operating system to run out of memory can cause consequences beyond just the compilation itself, possibly forcing the termination of other, unrelated processes.

One possible way to cause this issue in a Unix-like environment is to include the following line somewhere in the source code[2, Pg. 92].

#include "/dev/urandom"

Prevention  Attempting to prevent this kind of attack through static analysis is going to require a great deal of work, and may still only provide a patchwork coverage of the possible issues. On the other hand, by confining the entire build process to a sandbox one could place an overall limit on the amount of system resources that process may consume, without having to worry about enumerating every possible system call that may cause excessive resource consumption[30].

Denial of service: Consuming resources at runtime

Risk  It is easier to create a program that consumes resources endlessly at runtime than it is to create the same situation during compile-time, and it can be done no matter the programming language in question. The problems with
this are exactly the same as in attack described in 2.2.2, it may cause termination of other, important yet unrelated processes or inordinate disk swapping.

**Prevention** Prevention of this attack during runtime is similar to prevention of this attack at compile-time, place the process about to be run in a sandbox with appropriate limits upon the amount of system resources it may consume and terminate the process if that limit is exceeded.

### 2.3 Trust

Since the very purpose of this system is to provide material upon which decision on student grades may be formed, it is critical that the system itself can be considered trustworthy. By this I mean that the system can protect the integrity of the results, both the generation thereof during the test, and the result itself during and after the test, such that the assignment itself cannot alter the process. What considerations are important to keep in mind when establishing trust? What can the code being tested be allowed to do, and what should it be prevented from doing, and how is that best accomplished?

#### 2.3.1 Possible risks

In “Theoretical and Practical Aspects of Programming Contest Ratings”[2], the authors establish a number of attacks that a programming contest system may be the target of. A system for grading assignments is in many ways similar to such a contest. A high grade is desirable in both cases, but for that to happen the system has to function to the point that grading assignments is possible. Ill-will toward the system owner may cause a user to want to sabotage the system, hoping to cause problems for the system owners or other users. The situations classified as risks toward the trust of the system are listed below.

**Test subversion: Accessing restricted material**

**Risk** If there are materials such as the correct, expected answers for a test available somewhere, an attacker could attempt to access them during compile or runtime to aid in having his program produce the right result. Other targets could be the log files where the system writes the results of each test, where the attacker could append a note about a successful test. Some of these attacks only require read-access, while others also require write-access. Common among them all is that obscurity, or hiding the details of the system implementation, is not a sure-fire way to prevent this attack.

**Prevention** Prevention of this attack involves making sure the attacker has no access whatsoever to any of the sensitive materials of the system, neither during compile-time nor during runtime. Even an attacker with an in-depth
understanding of the system should be unable to access any of the important files in any way during the test.\[2, Pg. 92-93\]

**Test subversion: Misusing the network**

**Risk**  Access to the network is a potential source of much mischief. A user may be able to monitor other communication to and from their host, but also potentially to and from other hosts. This could include intercepting other solutions, passwords or other sensitive information. Taken one step further, an attack with a good understanding of the system could send traffic to communicate a passed test, if the system relies on network communication. Another possibility is for the user to establish a two-way communication between the testing system and himself, or the testing system and a third party, to explore the system for vulnerabilities or affect an attack to escalate privilege (see “Exploiting bugs in the operating system” in 2.3.1).

**Prevention**  The straightforward way to prevent this kind of attack is to ensure that the program being tested does not have access to the network in any way throughout the entire testing process. A test may involve network access in itself, in which case network access must be provided. In those instances it is important that any important network traffic related to the system is sent through encrypted channels to prevent eavesdropping. In addition the network’s hardware should be configured in such a way as to minimize the opportunity for intercepting traffic header for other hosts. The testing environment should also be limited to as little network access as is possible while still allowing the test the necessary resources, such as limiting the testing environment to local network access as opposed to full internet access.\[2, Pg. 93\]

**Denial of service/test subversion: Modifying or harming the testing environment**

**Risk**  There are indirect ways in which a program may consume resources, such as by allocating a very large file on the file system, or allocating very many small files, which can adversely affect the testing environment by causing a lack of disk space or exhausting the file-systems supply of inodes. Another possibility is for the attacker to alter the testing environment in creative ways, such as creating executable files in the local directory with the same filename as that of binaries used by the system, with the hopes that the system will execute the replacing binaries instead. Such a scenario would allow the attacker to replace parts of the AESEA system functionality with his own desired output.

**Prevention**  To prevent these kinds of attacks great care has to be taken when designing the sandbox environment, to disallow as many of these creative avenues of attacks as possible by making the environment as restrictive as possible. Additionally the system itself has to be carefully designed to make sure it does
CHAPTER 2. PRINCIPLES

not expose itself to the kinds of creative interference described above.[2, Pg. 93-94]

**Denial of service/test subversion: Circumventing the time measurement**

**Risk** CPU-time is a tempting measure for a program to determine success under a certain time constraint, or to determine for how long a program may execute before it is forcibly terminated. This kind of measuring can however be avoided by, for example, launching a child process to perform the demanding calculations, thus leaving that process with the burden of the CPU-time. The main process incurs no penalty as all it then needs to do is block while it waits for the child process to terminate, returning the result. This can also be used as an attack, by having the program attempt wait for an unavailable system resource or simply put itself to sleep, thereby incurring no CPU-time.

**Prevention** Preventing programs from creating additional threads would be a good way to prevent test subversion via additional processes, but would not prevent denial of service through other similar means. By coupling that approach with a real time limit for how long the program may execute or the test may carry on, the entire attack can be fully negated.[2, Pg. 94-95]

**Denial of service/test subversion: Exploiting bugs in the operating system**

**Risk** Whenever an attacker is given access to a system, as in the case of a program submitted for testing in the testing environment in this case, there is an opportunity to attempt to exploit bugs in the operating system. This is commonly done in order to elevate user privileges, often to gain super-user privileges. Successful elevation of privileges may allow the user to subvert the test by accessing the expected results or altering the test results. It can also be used to sabotage the test environment and make further testing impossible.

**Prevention** As the purpose of the AESEA system is to allow user submissions to be tested in a similar environment to that which they were developed in, this avenue of attack cannot be completely eliminated. Effort has to be taken to ensure that the environment that the tests are executed in does not expose itself to unnecessary risk or contain vulnerable executables.[2, Pg. 96] In addition, the system itself should be designed to avoid offering avenues of attack toward escalation, for example by applying privilege separation[37] where superuser privileges are necessary.

**Test subversion: Plagiarism**

**Risk** The automated nature of the testing makes this system vulnerable to plagiarism and solution sharing. While users are normally forbidden from shar-
CHAPTER 2. PRINCIPLES

...ing solutions and are instead expected to solve the assignments on their own, the system has no means of guaranteeing that is the case once the solutions are submitted.

**Prevention**  The automated nature of the system, along with the fact that all solutions will pass through the same system as they are tested, means that tools for code plagiarism detection\[35]\[24] can be put to use as one of the test steps. Care has to be taken that false positives generated by such a system, possible for assignments where each separate solution is likely to be small and similar to each other, are vetted by a human before being taken seriously. Steps also have to be taken to ensure that multiple submissions by the same user are not flagged as copies of each other.

**Obfuscation**

**Risk**  Obfuscation is in itself not an attack, but a means of hiding something in the program from a casual observer, such as a call to spawn a new process or an attack. A naive approach toward securing a system may implement a plain-text search of source code such as “grep”, to look for disallowed system calls. A more sophisticated approach could use static analysis (see “Static analysis of code” in 3.2.3) of program source code to ensure that it does not make use of any disallowed functions or system calls. The goal of obfuscated source code would be to prevent both of these approaches from detecting disallowed usages properly.[2, Pg. 96]

**Prevention**  Static analysis of code is not foolproof, in fact it is easily subverted with the right knowledge[12]. Security should therefore not rely on it, but rather on something more secure such as system call interpositioning (see “System call interpositioning” in 3.2.5). The less sophisticated approaches such as plain text searching are even less reliable than static analysis.
Chapter 3

Security alternatives

For this system to be considered robust and trusted (see trust principle, 2.3), it will have to handle both accidentally and intentionally malicious code submitted for testing in such a way that the system remains functional without major disruption and remains capable of producing correct, unaltered test results. However such security must not come at too great a cost relative to the performance and ease-of-use of the system.

This section will cover the available approaches for providing such security, along with the potential benefits and drawbacks they provide.

3.1 Implementations in existing systems

3.1.1 QUIMERA

QUIMERA does not make use of any kind of restricting sandbox during execution. Instead it attempts to block malicious software by defining a set of “suspicious” instructions[18], and disallows the execution of any program containing any of those or any system calls. This either limits the range of possible assignments the system can test (such as by disallowing file creation), or leaves holes in the security of the system (such as allowing file creation, which can then be used by an enterprising programmer to create and hide very large files in obscure places).

3.1.2 DOMjudge

DOMjudge changes which user the program is executed as and places the program it is testing in a chroot jail before execution[40]. This leaves DOMjudges sensitive files safe, while at the same time allowing the program being tested to create, read and write files. The process that performs this task normally needs special privileges to do these things, and this still leaves the system vulnerable to things such as network access.
3.1.3 Online Judge

Online Judge\cite{13} provides two different solutions.

The first solution offered is a sandbox module, meant to wrap single programs in a protective cocoon. This module works by hooking the system calls (see “System call interpositioning” in 3.2.5) the sandbox is configured to prevent, disallowing their use. In addition it also provides quotas for things such as CPU-time, wall-time, disk and RAM.

The second, more innovative solution, makes use of an external website, “ideone.com”\cite{4}, that allows users to provide code and have it compiled and executed on the remote server, which then presents the resulting output to the user. The service is also available via a proper webservice API. This leaves the entire system at the mercy of the Ideone service, both in terms of security and reliability, which is not ideal considering the high degree of reliability that is required from AESEA. There is also a limit to the number of submissions allowed per month, meaning either an ongoing expense to raise or void that limit, or issues with how many students may use the system per month, neither which is ideal.

3.2 Implementation alternatives

3.2.1 Solutions not considered

Since the focus of this report lies on the proposed system being developed for Linköping University, certain alternatives were not considered as they would not be practical within the existing constraints.

OS-based alternatives

A solution relying on a very specific operating system such as SELinux with its Mandatory Access Control\cite{39} or FreeBSD with its Jail-combination\cite{25} of sandbox and chroot would not work for the university as a whole, having many other requirements and needs to meet beyond just those of this system.

To use one of those operating systems just for testing with this system was also disregarded. Firstly, having proposed solutions developed in one operating system and then tested on a completely different one would be impractical and error-prone. Secondly, trying to keep two separate systems working in nearly exactly the same way, side by side, would be a maintenance nightmare.

Alternatives requiring invasive program changes

There are many alternatives that attack this problem by altering the way programs are written, such as mandating the use of a certain API or limiting implementations to a certain language. Since the purpose of AESEA is to test the knowledge of the programming languages in a realistic environment, under
the same circumstances they would normally be used in, such a solution would diverge from the overall purpose and as such are not considered.

3.2.2 No protection

This is the simplest possible solution, but also the most risky. It guarantees that programs will work as expected even within the system, since the programs being tested are executed within the same environment they were originally written in, with the same libraries available and the same compiler used. It also exposes the system to nearly every conceivable attack mentioned in 2.2.2 and 2.3.1. While users may be discouraged from any attempts at direct sabotage by the near-certainty of being personally held responsible, more subtle attacks meant to alter grades in a malicious way could easily go undetected.

Advantages

- Trivial to implement.
- Provides the highest possible performance in all aspects.
- Provides the highest possible compatibility.

Disadvantages

- No security provided whatsoever, requiring only minor insight to sabotage. Results may be outright falsified with more effort.

3.2.3 Static analysis of code

Since the system is provided with the raw source code for the program it may be possible to evaluate the code for malicious behavior through static analysis before it is executed. This analysis may be done on either the source code or the resulting binary executable[11].

In its simplest form, such an analysis might simply examine the code for any invocations of system calls to open files, fork new processes or other “undesirable” actions[11]. Analysis beyond that may allow more fine-grained control, but come at a high cost in terms of additional complexity. This task is made even more difficult by the need to support several different programming languages (see “Flexibility principle” in 2.1), each which may have its own idiosyncrasies regarding system calls.

The requirement for the system to work with many different programming languages introduces another difficulty, since the static analysis of one programming language is likely to be quite distinct from the same kind of analysis of another. One possible solution is to use a modular frontend for each language that generates a “middle-stage” data file to be used further during the static analysis[35][24], but the problem of having to implement the corresponding frontend for each supported language remains.
Finally, however well done the static analysis is, there remains the issue of what may happen at runtime. It is for example possible to imagine a C-program that, in its source code, creates a buffer of bytes containing lightly obfuscated binary code that in turn contain “forbidden” system calls, and the program itself then deobfuscates that binary code and executes it. A program can also load an external library or otherwise gain the means to call the “forbidden” system calls, in ways that are hard or impossible\cite{12} for a static analysis to uncover.

**Advantages**

- Provides a assessment as to whether the program is safe before it is executed.

**Disadvantages**

- Time-consuming to implement for many different programming languages.
- Complex to implement properly, and security relies on flawless implementation.
- Cannot provide a fully accurate assessment of the program’s safety.
- Requires regular maintenance to keep up with evolving programming languages.

### 3.2.4 Using a custom chroot solution

A logical step up from no protection is a chroot\cite{31} solution to mount or otherwise export the files and directories about to be used into a separate file-environment. A chroot-jail can be created in that particular file-environment, followed by changing the current user to something safe and benign before handing over control to the program being tested. This limits the programs access to the file-system, protecting sensitive information and preventing the use of unwanted executables. However, networking and CPU/RAM resources are still made available, and if modified, “sanitized” versions of files or directories have to be made available the setup involved in copying files can be a burden\cite{36}. A program that is able to gain superuser privileges may also be able to escape the chroot.

More sophisticated tools such as the File Monitoring and Access Control (FMAC)\cite{36} may alleviate this issue. FMAC provides a custom file-system based on the existing base file-system, with certain files and directories hidden or made available with reduced permissions.

**Advantages**

- Isolates the program from sensitive files and directories with a minimum of runtime overhead.
- Works unchanged across a broad range of programming languages.
Disadvantages

- Does not prevent the program from accessing network, CPU or RAM resources in any way.
- Must prevent the program from gaining superuser privileges to remain effective.

3.2.5 System call interpositioning

Related to the static analysis of code approach, the interpositioning approach works by placing itself between the program in question and system calls that need some sort of access control, often including calls such as fopen, fork and socket. These system calls are then be transparently intercepted, altered or denied[21]. This can be accomplished through a number of different means, each with its own set of advantages and disadvantages.

Interpositioning through custom libraries

A basic approach to system call interpositioning is provided by loading custom dynamic libraries into the program that in turn intercepts the desired system calls. Used by systems such as COLA[26], this approach is lightweight and requires no specific kernel support or program changes, all that needs to happen is for the program to be relinked with the appropriate replacement dynamic library. On the other hand this approach is also easily bypassed, by for example a statically linked executable. Since some programming languages will execute through their own virtual machine (such as Java), through a binary which cannot be directly influenced, that means that this approach would not be useful for all of the targeted programming languages.

Kernel based interpositioning

A more powerful approach to system call interpositioning is be accomplished by implementing it with kernel support.

By assigning a wrapper[20] to some function calls of a certain process, the kernel can then make sure that the wrapper is called prior to the evaluation of the actual system call. This wrapper may then perform additional checks before letting the system call proceed, or possibly even reject the system call entirely.

Not only does a common interface for lightweight wrappers, as defined by “Hardening COTS software with generic software wrappers”[20] significantly ease development, but the implementation of it through the kernel ensures that avoiding the interpositioning layer is impossible[32].

Process-tracing interpositioning

Many operating systems make process-tracing facilities available for debugging purposes. System calls such as trace, strace or truss, along with system calls
like ptrace to assign a callback, can be used to monitor when the traced process makes a system call, and what system call it is\textsuperscript{[22]}. Unfortunately this kind of tracing tends to be very coarse, either all system calls are traced, or none are. Similarly many operating systems also do not offer any way to deny a system call, other than completely killing the traced process. This may cause issues with programs run by, for example, the Java VM, which may make system calls beyond control of the user.

### Advantages

- Allows for the detection and blocking of potentially unsafe system calls while a program is running, before it can do any harm.
- Works unchanged across a broad range of programming languages.

### Disadvantages

- Requires support from the operating system.
- Does not prevent misuse of CPU resources.

#### 3.2.6 Using a virtual machine

A virtual machine provides an extended sandbox, providing everything one would expect from an operating system running directly on a host machine, with the added advantage that no matter how subverted the operating system inside the virtual machine becomes, any malicious code has to break out of the virtual machine itself before it can do any actual damage to the “real” system. This adds another layer of security with minimal exposure to attack.

There are many different virtual machine solutions available, so to thin the herd I am adding a couple of constraints on which virtual machines will be considered:

- **Shared directories** This features allows the host operating system to share a directory with the guest operating system running on the virtual machine, making importing and exporting files related to the test easy to accomplish. No difference is made between read-only and read/write shared directories for this comparison.

- **Snapshots** This feature lets one save the entire state of a virtual machine at a certain point in time to disk, to later be restored. Additionally the virtual machine can later be rolled back to this snapshotted state, undoing any changes to the system that has occurred since. This makes it easy to reuse the same instance of a virtual machine over and over again, without changes from previous tests affecting the next.
User Mode Linux

User Mode Linux[8] virtual machines requires Linux as the host operating system, and provides a virtual machine that can run another installation of Linux. Other operating systems than Linux are not supported by User Mode Linux, and the host operating system has to run either on a fairly recent Linux kernel, 2.6.0 or better, or a lower version with the appropriate patch installed. It also does not support emulating a different guest CPU than that on the host machine. That said, it does support shared directories and snapshots of virtual machine state, and it is both open source software and made available free of charge, but also without the official support of a backing company.

VirtualBox

VirtualBox[7] is a virtual machine available for a broad spectrum of operating systems, ranging from Windows to Mac OS X, Linux and Solaris, and allows for guest operating systems across the same range and more. It is limited to emulating x86-architecture CPUs in the virtual machine, however. It is provided by Oracle, and is available both free of charge as open source under a GPL 2 license, and as a proprietary solution under a commercial license with added enterprise features. It supports all the basic features expected from a modern virtual machine, including snapshots and shared directories, but also many sophisticated features such as 3D acceleration.

VMware Workstation

The VMware Workstation[9] virtual machine is available for Windows and Linux as host operating system, while it supports a much wider range including Windows, Mac OS X, Linux and Solaris as guest operating systems. It is limited to emulating x86-architecture CPUs in the virtual machine. It is sold by VMware and is a fully proprietary solution, not available as open source. Its features include all the basic features AESEA would need, including snapshots and shared directories, as well as more complex features such as 3D acceleration.

Advantages and disadvantages of virtual machines

Advantages

• Very secure against most kinds of attacks.

• Does not require extensive expertise to configure and run safely.

• Allows saving machine state for later analysis, in case of errors or malicious code.

Disadvantages

• Heavy resource consumption.
• The configuration of the virtual machine must be kept up to date with the configuration of the “real” machines.

3.3 Conclusion

With the risks of running unvetted program code without protection being as they are, potentially disturbing the examination of hundreds of students, there is little reason to not implement some basic protective measures. For example a solution that transfers files into a chroot, runs them with precautions such as implementing a quota on CPU-time, wall-time, RAM and disk, and then copies them back out again. Such a solution would go a long way toward protecting against the most devastating of attacks, while at the same time being possible to implement without too much effort, and for most operating systems that AESEA is likely to be run on.

To take the protection one step further and defend against attacks involving things such as the network, some kind of third-party interpositioning solution would be appropriate. It is one of the more complex solutions, but it is also lightweight and powerful, once properly implemented. To ensure that there are no ways around the interpositioning a kernel-based solution (see “Kernel based interpositioning” in 3.2.5) would be the best choice. Since network access is required for some assignments, such a solution would also need to be configurable.

If system call interpositioning is not available or practical, the fall-back solution would be to use a virtual machine. This allows for the testing of programs running on different guest operating systems than the host operating system, as well as providing maximum insulation and protection against potentially malicious programs.

One downside is that running one or several virtual machines will put a heavier strain on the host machine compared to running the same number of chrooted programs. Another downside is the additional upkeep burden somebody has to bear in keeping these guest operating systems up to date with the environment that the programs being tested on them require.

In this case, which virtual machine software is the most appropriate to use would weight heavily upon maintenance “cost”. In other words, how time-consuming it is to maintain and keep the virtual machine current with the “real” system it mirrors is a major deciding factor. This ties back to the robustness principle described in 2.2. If no constraints are placed upon the choice of virtual machine, User Mode Linux would be our first choice due to its low complexity and low resource use, with VirtualBox the second choice due to its wide range of supported systems coupled with its low cost.

Static code analysis as a way to secure against malicious code just is not practical in any real sense[12], and that coupled with the troubles of implementing it properly for several programming languages at once, makes it wholly unsuitable for this task.
Chapter 4

Configuration format

Since the system requires a great deal of flexibility (see “Flexibility principle” in 2.1), it is important that the configuration files for the system are able to fully define the behavior of the system for each particular case. At the same time the configuration format has to be easy for a user to read and write (see “Robustness principle” in 2.2). A large part of the user interaction with the system is going to be in writing and modifying these configuration files, so a great deal of thought has to go into choosing and designing an appropriate format for these configuration files.

This section will cover some of the more prominent and notable configuration formats available. There are other formats beyond those that are covered here, but they are either outright unsuitable or niche to the point where they would be impractical to use.

4.1 Formats in existing systems

4.1.1 QUIMERA

QUIMERA[18] makes use of a web-based configuration interface. The amount of configuration options is limited by the somewhat fixed nature of the system. Options that can be configured includes whether a test is “public” or only visible to administrators, the maximum allowed program execution time, inputs and respective expected testing outputs for each test. There are no actual configuration files for tests, instead the configured values are saved directly to a database server.

4.1.2 DOMjudge

The configuration of DOMjudge[40] is scattered across several different places. Some configuration options are accessible through a web-based interface, other settings need to be set directly in PHP-files using that particular language, while others yet, such as how programs should be compiled, executed and the
expected results of each execution produced, are configured through the writing of custom shell-scripts. There is therefore no overarching configuration format.

4.1.3 Online Judge

There is precious little information available about the configuration possibilities for Online Judge[13]. What little I can find indicates that a few minor configuration options such as starting and stopping tests are available through a web-based interface, while the meat of the configuration is done by editing PHP-files and shell-scripts responsible for executing the different tasks. As such no specific configuration format can be said to exist.

4.2 Important features

4.2.1 Readability

In keeping with the robustness-principle (see 2.2), the configuration format has to be easily readable by humans. Since tests for one exam will be different from the tests for the next, the editing of configuration tests are going to recurring activity in the use of this system. To keep the system actively used, it is therefore important that the configuration format is easy to use and understand, such that the people editing are not discouraged by obscure syntax or cryptic keywords, and no specialist knowledge is required. What options are being configured, for what module, to what values, has to be clear at a glance.

Related to this is being able to add comments, ideally the configuration format will support in-line and/or block comments. These will help with labeling and clarifying, which will be invaluable for any configuration files that see use or reuse over a longer period of time.

4.2.2 Nesting

An exam normally consists of several assignments, each with its own set of AESEA tests. To prevent repetition of the same configuration-options for each and every individual test in the configuration files, it should be possible to specify “default” options in one place, that may then be overridden with specific options for individual tests. Another consideration is that when the individual tests are performed, those tests have to be provided with a configuration file focusing only on their options. This means that there has to be a way to specify which configuration options are for what set of tests, and for what module in each individual test.

All this could be easily accomplished, if the configuration format in question supported nesting blocks or scoping, preferably named blocks and scopes. Being able to define “global” values and later override them with “local” values would also be beneficial.
This feature is directly tied to the flexibility principle (see 2.1), the configuration format has to provide the ability to fully define each individual option for tests, while at the same time being flexible enough not to require it.

4.2.3 Scriptability
With automated testing it is easy to perform many similar tests with just minute variations of the input data or test conditions, to exhaustively test a particular aspect of a program. If each of those tests have to be manually configured and added, making full use of this possibility becomes a onerous task that either consumes precious time or leaves this added benefit not fully utilized. To alleviate this issue, it would be very beneficial if parts of the configuration format could also allow some form of scriptability, such as repeating one test N number of times, each time with just a small variation in one particular option.

This feature also relates directly to the flexibility-principle (see 2.1), the easier the configuration format makes it for us to re-use different variations of the same configuration without having to rewrite the whole thing, the likelier the user will be to make full use of the systems capabilities in that regard.

4.2.4 Serializability
Related to the nesting-feature, the modular design of the system (see “Modularization” in 2.2.1) will require the ability to specify configuration options for individual modules, either default values or specific values for each test. That in turn requires the ability to read and parse a configuration file, then output another, refined version containing only the relevant configuration for a particular test or module. In other words, not only does the system in question have to be able to read and understand the configuration format, it also has to be able to translate a modified set of settings back into the configuration format again.

This feature also relates to the robustness-principle (see 2.2), the system needs to be able to parse the configuration and re-write it back to disk without this process altering the content of the configuration in any meaningful way.

4.3 Existing configuration formats

4.3.1 YAML
YAML 1.2[1], or “YAML Ain’t Markup Language” version 1.2, is described as “a human friendly data serialization standard for all programming languages”. Originally specified in 2001, it is one of the more popular configuration formats available, with bindings available for many programming languages.

It is based on three basic primitives, dictionaries, lists and scalar values, which are more than enough to describe most data structures.
Example

```
# YAML test configuration example
defaults:
    sequencer:
        tests: { test 1, test 2 }
        allow parallel tests: false
terminal:
    timeout: 3
tests:
    test 1:
        files: { file1.txt, file2.txt }
        terminator:
            timeout: 60
    test 2:
        files:
            - file1.txt
```

Readability

YAML-files require very few indicators in relation to the data it contains, leaving the data front and center. Comments are easily embedded in the rest of the structure.

YAML also allows for a great deal of flexibility in how multi-line text is handled. Multiple lines of text can be written as such in the configuration file, and will be treated that way. An example of multi-line text is provided later, in 5.2.1. By using special markup it is also possible to write a singular long line as several lines in the configuration, where each linebreak then ends up replaced with a single space-character, resulting in one unbroken line when the YAML-file is parsed.

Overall an eminently readable format.

Nesting

YAML handles nested data very well. It allows for the use of indentation to indicate scope, but scope indicators can also be used. It also allows data nodes to be defined in one place, then referenced in another without having to re-enter them fully, akin to “using pointers”. YAML calls these “aliases”[10].

Scriptability

YAML does not support any kind of scripting. Support would have to be added separately, perhaps as a module to parse the configuration file, interpret special options and output a “compiled” version of the configuration file to disk as a result.
Serializability

YAML is a well-defined language that has been around since 2001, and there are many third-party libraries for reading from and saving to YAML-files available.

4.3.2 JSON

JSON[5], or Javascript Object Notation, is a lightweight data-interchange format based on a subset of the Javascript programming language. Originally specified somewhere around 2001, it is a widely used format for data serialization on the world wide web, especially in combination with Ajax techniques.

JSON is based on dictionaries, lists and scalar values.

Since JSON is a subset of Javascript, it is sometimes convenient to extend JSON to full Javascript to achieve things such as scriptability, and as such JSON and Javascript will used interchangeably, depending on the context of the discussion.

Example

Please note that the example below is nearly identical to the examples of Python and Ruby.

```json
{
   "defaults": {
      "sequencer": {
         "tests": [ "test 1", "test 2" ],
         "allow parallel tests": false
      },
      "terminator": {
         "timeout": 3
      }
   },
   "tests": {
      "test 1": {
         "files": [ "file1.txt", "file2.txt" ],
         "terminator": {
            "timeout": 60
         }
      },
      "test 2": {
         "files": [
            "file1.txt"
         ]
      }
   }
}
```
Readability

JSON is relatively indicator-heavy, as can be seen above, which hurts the readability. Whitespace for formatting is voluntary, but comments are not supported unless embedded as a special form of data that then has to be explicitly ignored when handling the data. The syntax hurts readability overall.

One special issue of note, JSON does not support larger sections of text with linebreaks embedded, they have to be escaped as \n instead. An example of this is provided later, in 5.2.1.

Nesting

JSON has no issues with nested data. The syntax describes the scope clearly, scope delimiters are mandatory and if whitespace is used to “indent” the scoping becomes quite clear.

Scriptability

JSON has no explicit support for any kind of scripting. One could easily imagine extensions to a configuration format based on JSON using a Javascript script engine for interpretation, which could in turn produce a JSON-formatted result. That would have to be implemented above and beyond existing JSON reading/writing libraries, however.

Serializability

JSON is a popular format data serialization, and as such there are third-party libraries for reading from and writing to JSON for most programming languages, all the way from Cobol to C#.

4.3.3 XML

XML 1.0[43], or Extensible Markup Language version 1.0 fifth edition, is a flexible format that is easily customized for most situations. Originally drafted somewhere around 1996, it has been around for a long time and remains a popular language with many applications. XML uses external schemas which define what elements are allowed and in what combination, meaning that one can take this very flexible format and define and restrict it to something very well-defined.

XML is a loose set of rules rather than a collection of datatypes, but can easily be used to represent dictionaries, lists and scalar values as needed.

Example

<!-- XML test configuration example -->
<configuration>
<defaults>
CHAPTER 4. CONFIGURATION FORMAT

<sequencer>
  <scheduled_tests>
    <scheduled_test>test 1</scheduled_test>
    <scheduled_test>test 2</scheduled_test>
  </scheduled_tests>
  <allow_parallel_tests>false</allow_parallel_tests>
  <terminator>
    <timeout>3</timeout>
  </terminator>
</sequencer>
<defaults>
<tests>
  <test>
    <name>test 1</name>
    <files>
      <file>file1.txt</file>
      <file>file2.txt</file>
    </files>
    <terminator>
      <timeout>60</timeout>
    </terminator>
  </test>
  <test>
    <name>test 2</name>
    <files><file>file1.txt</file></files>
  </test>
</tests>
</configuration>

Readability

XML is extremely verbose, but in a way that hurts rather than helps readability for a human reader. As can be seen above, XML with roughly the same content as the YAML or JSON formats becomes both longer and “noisier” to the eye. Comments are supported as embedded single tags, but that is not enough to save the readability of the format.

Nesting

XML nests data just fine, as long as the defined schema allows for it. With whitespace optionally used to “indent”, as seen above, the depth of the nesting is clear.

Scriptability

XML has no explicit support for any kind of scripting, some sort of special elements would have to be defined and then parsed through an external program
instead to allow for it.

An alternative exists in the form of XSL transformations[44], but as that implementation is outside the XML-standard is it not examined further.

Serializability

XML is a widely-used data format with many years of use, and as such there are countless third-party libraries to choose from. As it cannot be directly tied to an in-memory data structure in the same way as for example JSON, reading from and writing to XML is a little more involved than the other formats, which can become cumbersome for a system such as AESEA where many different modules all need to read and understand the same format.

4.3.4 INI

INI, or .CFG, .conf, .txt, is not a strictly defined format like the others covered here, yet it is a common lightweight alternative for cases where not much complexity is needed. With no formal definition the format comes in many shapes and forms, with features as varied as the number of different implementations in use. I will be covering a makeshift format similar to what one might see, but not adhering to any one official standard, with the adaptations needed to specify default options and separate tests.

INI-files traditionally contain key-value pairs defining attributes, sometimes divided into named sections.

Example

; INI test configuration example
# YAML test configuration example
[defaults]
sequencer-tests=test_1:test_2
sequencer-allow_parallel_tests=false
terminator-timeout=3
[tests]
test_1-files=file1.txt:file2.txt
test_1-terminator-timeout=60
test_2-files=file1.txt

Readability

INI-files can often end up being fairly short, as seen above, but the inability for the format to represent more complex data structures hurts readability as such meta-information instead has to be embedded in the key-names. Since the format is not strictly defined, that leaves the formal syntax of the configuration in limbo and further hurts readability.

One can easily implement full-file comments, but overall the INI-format is not very readable.
Nesting
INI-files do not support nesting sections as sections do not have any formal “end-tags”, which means that nesting instead has to be expressed through the key-names, as seen above. As the nesting depth increases this quickly makes this format unwieldy.

Scriptability
There is no support for scripting of any kind for INI-files, and of all formats covered here INI would be most difficult format for which to implement an external program solution that provides this.

Serializability
Being very loosely defined with widely varying interpretations and implementations, serializing to INI from one library or implementation then attempting to de-serialize with another is risky and will require in-depth testing to make sure no issues arise. There exists libraries one can use, but since there is no actual INI-standard available one cannot expect the same behavior from libraries for different language platforms, or even different libraries for the same language platform.

4.3.5 Lua
Lua 5.2[41] is “an extension programming language designed to support general procedural programming with data description facilities”. It is a lightweight scripting language meant to be embedded into other programs, and as such differs from some of the other formats described here in that it is more than just a data format. Being a fully fledged scripting language means that it is heavier to embed than just a simple data loading/saving library, but at the same time also provides a more extensive feature set.

Lua supports structures similar to dictionaries and lists, and provides the usual scalar values.

Example
-- Lua test configuration example
configuration = {
  defaults = {
    sequencer = {
      tests = { "test 1", "test 2" },
      allow_parallel_tests = false
    },
    terminator = {
      timeout = 3
    }
  }
}
Readability
Lua has many similarities to JSON in that it is fairly indicator-heavy, whitespace may be used for formatting but are not required. Unlike JSON Lua supports comments, both line-comments and block-comments. A couple of special cases such as having to use a special syntax when using a string containing spaces as key leaves the readability not ideal, but with comments it remains a decent choice.

Nesting
Lua nests to arbitrary depths without any issues, and so long as whitespace is used to format it is clear what exists in what scope.

Scriptability
Lua is different from some of the other languages in that it is a fully-fledged scripting language, and as such one can easily embed code in the configuration file to, for example, generate a more complex and comprehensive set of actual tests based on a smaller “seed”-configuration.

Serializability
Despite its roots as a data storage format, Lua does not have any proper serialization/de-serialization to/from text features built-in. Such a feature would have to be implemented separately. One can easily load an entire configuration file and execute it as a Lua-script however, and whatever data it contains in the proper Lua-format would end up loaded. This would also execute any other Lua-code that the configuration file contains, for better and worse. That might be considered a feature, but also a security risk if the configuration files are not trusted.
4.3.6 Python

Python 3.2\cite{2008} is a dynamic language designed to emphasize readability. It is often used as a scripting language and can be embedded into other programming languages. Since the latest major release of Python, Python 3, is not fully backward-compatible with Python 2, I will focus solely on Python 3 in this discussion with the assumption that as time goes on it will become the more prevalent version of the two. Python can be embedded into programs written in other programming languages easily.

As a fully-fledged programming language Python supports all of the basic data structures, including dictionaries, lists and scalar values.

Example

Please note that the syntax below is nearly identical to the examples of JSON and Ruby.

```python
# Python test configuration example
{
    "defaults": {
        "sequencer": {
            "tests": [ "test 1", "test 2" ],
            "allow parallel tests": False
        },
        "terminator": {
            "timeout": 3
        }
    },
    "tests": {
        "test 1": {
            "files": [ "file1.txt", "file2.txt" ],
            "terminator": {
                "timeout": 60
            }
        },
        "test 2": {
            "files": [ "file1.txt"
        ]
    }
}
```

Readability

Like JSON, Python is indicator-heavy, which hurts readability. Whitespace formatting is optional for pure data. Unlike JSON, Python supports embedding
comments, which makes it much easier to read. Python also supports larger sections of text with linebreaks embedded without any issues, unlike JSON. All of this adds up to a reasonably readable format.

**Nesting**

Python, like JSON, has no issues with nested data. The syntax describes the scope clearly, and if whitespace is used to “indent” the scoping becomes quite clear.

**Scriptability**

As Python is a fully-fledged programming language, functions and code could be embedded in the configuration files to, for example, generate a more complex and comprehensive set of actual tests based on a smaller “seed”-configuration.

**Serializability**

Python is not written with the express intent of providing a good data storage format, and as such it will take a little work to make serialization to disk as plain text, but it is eminently doable. De-serialization is simple, as long as the configuration file is trusted not to contain malicious code it could simply be “executed” within the Python-context to regain whatever variables were stored within.

### 4.3.7 Ruby

Ruby 1.9[42] is dynamic, reflective, object-oriented interpreted language created in 1995. It has extensive meta-programming possibilities, which are often used to provide new features with more convenient syntax than more rigid programming languages would require for the same functionality. Embedding Ruby takes more work than simply using a library for one of the more focused data storage formats, and is likely to be a problem in a system where many of the modules are written in different programming languages.

Like most programming languages Ruby supports associative arrays, which for all intents and purposes act as dictionaries, as well as other lists and scalar values.

**Example**

Please note that the syntax below is nearly identical to the examples of JSON and Python.

```ruby
# Ruby test configuration example
$configuration = {
  "defaults" => {
    "sequencer" => {
      ...
```
"tests" => [ "test 1", "test 2" ],
  "allow parallel tests" => false
},
"terminator" => {
  "timeout" => 3
}
},
"tests" => {
  "test 1" => {
    "files" => [ "file1.txt", "file2.txt" ],
    "terminator" => {
      "timeout" => 60
    }
  },
  "test 2" => {
    "files" => [ "file1.txt" ]
  }
}
}

Readability

Nearly identical to Python in all aspects, Ruby is indicator-heavy and white-space formatting is optional. Ruby supports embedding comments, as well as larger sections of text with linebreaks embedded. All this makes Ruby a reasonably readable format, on equal footing with Python.

Nesting

Like the other programming languages Ruby has no issues with nesting. The syntax describes scope clearly, and if whitespace is used to “indent” the various scopes are easy to see.

Scriptability

As with the other programming languages covered, functions and code can easily be written along with variables in the configuration files to extend behavior to, for example, some sort of test-generator.

Serializable

De-serialization of saved Ruby variables and code is easy to accomplish, as long as the configuration is considered trusted it can simply be evaluated. Serializing Ruby back to plain text is a challenge that requires some work or external libraries, however.
4.4 Conclusion

Looking at these possible configuration formats from the perspective of nesting, the INI-format can be almost immediately discarded. It simply does not support a required key feature that all other formats provide.

From a readability-standpoint both the INI-format and XML are lackluster. It needs to be possible to come back to certain a configuration file a year or more since it was last touched, and still be able to quickly ascertain what it did and what modules it involved. The extra “noise” these two formats introduce make them quite undesirable in that regard. YAML comes out far ahead of the other formats, with its light and flexible syntax.

As far as scriptability goes Lua, Python and Ruby are the strongest contenders, but JSON is not far behind as embedding a full Javascript scripting engine should not be much different from embedding a Lua scripting engine in the first place. If such a “JSON+”-parser is written as a separate module and normal JSON-reading libraries used in the rest of the modules of this system, the JSON-approach might even be simpler than the Lua alternative. Additionally, adding a Javascript-step before the actual, final JSON output is generated also makes it possible to include Javascript-style comments in the initial, hybrid configuration, negating that downside to the JSON format as well.

Finally, as for serializability the weakest contender is, first and foremost, the INI-format. With its lack of a formal specification its very likely that different reading/writing libraries will do things slightly differently, enough to introduce compatibility issues. Next up is Lua and Ruby, not having a “built-in” serialization/de-serialization feature means each module will have to implement its own, introducing another possible source of bugs and maintenance woes. Python is easier to serialize to plain text than Lua and Ruby, but Lua and Ruby beat the remainder in this regard.

With all these factors in mind, JSON, Python and Ruby stand out as the best candidates overall. They all have their own individual challenges, strengths and weaknesses. However, one important consideration that has yet to be taken into account is user familiarity with the format chosen, and while Python and Ruby both stand on roughly equal footing, ability-wise, the system will be used in an environment where a lot of Python know-how is available, but not so much for Ruby. With this taken into consideration, the candidates for further investigation are reduced to JSON and Python.
Chapter 5

Result format

The output from the individual tests has to be human-readable in the end, but also machine-readable to some extent to allow modules to refine and collate the resulting information, distilling and presenting only the interesting information to the end-user. The result format shares a few feature requirements with the configuration format, but is slightly different due to the contents of the result files.

5.1 Important features

5.1.1 Readability

Readability is important, but to a lesser degree in the result format than in the configuration format, due to the fact that the result will often be large chunks of text in relation to the markup wrapped around the text. In other words it will not matter much if the syntax around that text is a little bit hard to read, since there will not be much of it in place.

Comments will not be necessary in the result format.

5.1.2 Nesting

The result format will also require the ability to nest output, since several consecutive modules may want to add their own data to the result for each test. For example, the executor module[28] may add the command executed to the output, which is then handed over to the terminator which in turn adds the command line it used, along with the amount of time taken and the peak RAM usage, for example. To easily be able to sort out what module added what information, a nesting format is invaluable.
5.1.3 Serializability

Similar to the serializability argument for the configuration format, the result format needs to be possible to read, parse, add more information to, then serialize back to disk in the same format without altering the existing content.

5.2 Example result format

Rather than repeat the discussion for each of the important features mentioned above, from the configuration format section, in this section again, I will just refer to “Existing configuration formats” in 4.3. Instead I will here provide examples of how the result format file might look, given each format.

5.2.1 YAML

```yaml
executor:
  terminator:
    command: >
      executor --conf executor-conf.json --command "input-source --conf input-conf.json | myProgram ./inputdata.txt | output-comparator --conf output-conf.json"
    command-performed: >
      input-source --conf input-conf.json | myProgram ./inputdata.txt | output-comparator --conf output-conf.json
  wall-time: 1.74
input-source:
  command: input-source --conf input-conf.json
  input: |
    start
    start
    end
output-comparator:
  result: PASS
  command: output-comparator --conf output-conf.json
  output: |
    Input choice:
    The file contains the following names:
    Jonas
    Lindgren
    Input choice:
    The file contains the following names:
    Jonas
    Lindgren
    Input choice:
    Quitting..
```
CHAPTER 5. RESULT FORMAT

JSON

Note that some lines below were abbreviated to fit the page width.

{
    "executor": {
        "terminator": {
            "command": "executor --conf executor-conf.json --command...",
            "command-performed": "input-source --conf input-conf.json...",
            "wall-time": 1.74
        },
        "input-source": {
            "command": "input-source --conf input-conf.json",
            "input": "start
start
end"
        },
        "output-comparator": {
            "result": "PASS",
            "command": "output-comparator --conf output-conf.json",
            "output": "Input choice:
The file contains the following...
The file contains the following names:
Jonas
Lindgren
        }
    }
}

5.2.2 XML

Note that some lines below were abbreviated to fit the page width.

<executor>
    <terminator>
        <command>executor --conf executor-conf.json...</command>
        <command-performed>input-source --conf ...</command-performed>
        <wall-time>1.74<wall-time>
    </terminator>
    <input-source>
        <command>input-source --conf input-conf.json</command>
        <input>start
start
end
        </input>
    </input-source>
    <output-comparator>
        <result>PASS</result>
        <command>output-comparator --conf output-conf.json</command>
        <output>Input choice:
The file contains the following names:
Jonas
Lindgren
        </output>
    </output-comparator>
</executor>
**5.2.3 INI**

This format is so unsuitable for this use-case that I simply refer to the previous example in “Configuration format”.

**5.2.4 Lua**

Note that some lines below were abbreviated to fit the page width.

```lua
executor = {
    terminator = {
        command = "executor --conf executor-conf.json --command \"...\",
        command-performed = "input-source --conf input-conf.json | ...",
        wall-time = 1.74
    },
    input-source = {
        command = "input-source --conf input-conf.json",
        input = "start
start
end",
    },
    output-comparator = {
        result = "PASS",
        command = "output-comparator --conf output-conf.json",
        output = "Input choice:
The file contains the following names:
Jonas
Lindgren
Input choice:
The file contains the following names:
Jonas
Lindgren
Input choice:
Quitting.."
    }
}
```

**5.2.5 Python**

Note that some lines below were abbreviated to fit the page width.
{ "executor": { "terminator": { "command": 'executor --conf executor-conf.json --command "...", 'command-performed": "input-source --conf input-conf.json | ...", "wall-time": 1.74 }, "input-source": { "command": "input-source --conf input-conf.json", "input": ("start"start end""" ), }, "output-comparator": { "result": "PASS", "command": "output-comparator --conf output-conf.json", "output": (""""""""""""""); The file contains the following names: JonasLindgrenInput choice: The file contains the following names: JonasLindgrenInput choice: Quitting.."""" } }, } } 

5.2.6 Ruby

Note that some lines below were abbreviated to fit the page width.

{ "terminator" => { "command" => 'executor --conf executor-conf.json --command "...", "command-performed" => "input-source --conf input-conf.json | ...", "wall-time" => 1.74 }, "input-source" => { "command" => "input-source --conf input-conf.json", "input" => "startstart end", """"""""""""""); } } }
CHAPTER 5. RESULT FORMAT

43

```
},
"output-comparator" => {
"result" => "PASS",
"command" => "output-comparator --conf output-conf.json",
"output" =>
"Input choice:
The file contains the following names:
Jonas
Lindgren
Input choice:
The file contains the following names:
Jonas
Lindgren
Input choice:
Quitting..
}
```

5.3 Conclusion

Looking back at the possible formats available for configuration files with these new restrictions in mind, the INI-format is again immediately discarded, due to the lack of nesting.

Looking at things from a readability-standpoint, the XML-format is no longer an unattractive alternative. The relatively large chunks of text the output format will contain, along with the fairly restricted and pre-defined possibilities for output types, makes XML fairly readable. YAML is also eminently readable, but JSON suffers greatly from its inability to allow linebreaks embedded in longer paragraphs of text in any other way than as escaped characters. Trying to read a “raw” JSON result file, containing multi-line output, will prove an exercise in frustration for the reader.

As far as serializability goes, the INI-format, Lua and Ruby are both among the weakest candidates followed closely by Python, in line with the discussion regarding the configuration format above. XML, YAML and JSON all provide excellent serialization and de-serialization.

Just considering the result format in itself, YAML is the best candidate by a good margin due to its straightforward syntax and the flexibility in how it deals with multiple lines of text. However, if a different format is used as the configuration format, using the configuration format for the result format as well would be advisable. Minimizing the amount of different libraries involved in each module reduces the potential for bugs, which is desirable according to the robustness principle (see “Robustness principle” in 2.2). With that in mind, our recommendation for result format is whatever format was chosen for the configuration format, and if no choice was made there, YAML would be our secondary recommendation for the result format.

A special note regarding JSON, if it is to be used it would be advisable
to only use it as an intermediary result format, for parsing and handling the output programatically, before finally applying an output-step that transforms the result to a conveniently human-readable raw text format.
Chapter 6

Python vs JSON, in-depth comparison

To reach a decision regarding which format should be used in the proposed system I am comparing the two most promising candidates, Python and JSON, more in-depth. The primary focus of this comparison will be on their use as a configuration format, since that part of the system is going to require constant interaction to update tests and prepare for exams.

6.1 Python as scriptable configuration format

A first step toward comparing Python and JSON is to manufacture a reasonably realistic test-scenario. Toward that end I wrote a simple configuration loader using a Python shell-script (see A.1) and used it to load two different configurations. The first configuration is an example of an, albeit simplistic, fully defined test configuration, containing input and expected output values to the GNU “bc” command-line calculator[3]. The second configuration represents the same test-configuration, but in a shorter, more easily extensible format where the actual configuration is generated by a short snippet of Python code, rather than fully defined.

6.1.1 Fully defined configuration

```
# Python test configuration example
def config_generator():
    config = {
        "tests": {
            "numeric test 1": {
                "input": "1+2\n1*2",
                "output": "3\n2",
            },
            "numeric test 2": {
```

```
Clear and readable, but verbose, defining tests in this way is clearly useful for few, isolated instances, but when there are many similar, yet subtly different tests to be perform, as in this example, having to write each one out will quickly become unwieldy.

It is worth to note that I could here, instead of using “\n” to denote line-breaks, include actual linebreaks in the code, with the same result (see “Python readability” in 4.3.7). In this case not doing so helps readability more, but in instances where more lines of input or output needs to be added, using linebreaks might be preferable.

### 6.1.2 Programmatically defined configuration

# Python test configuration example
def config_generator():
    config = {
        "tests": {
        }
    }
    # generate some numeric tests
    count = 0
    for i in range(1, 10):
        count = 1 + count
        test = {
            "input": "\(\text{\{0\}+\text{\{1\}}\backslash n\text{\{0\}}*\text{\{1\}}\)\).format(i, i+1),
            "output": "\(\text{\{0\}+\text{\{1\}}\backslash n\text{\{0\}}*\text{\{1\}}\)\).format(i+i+1, i*(i+1)),
        }
        config["tests"]["numeric test \{0\}".format(count)] = test
    return config

Knowing that this configuration generates the equivalent to the one above, this one is in our opinion much more readable and easy to work with. I could easily extend this one to test a larger range of numbers, or add subtraction and division to the tests as well.

### 6.2 JSON (Javascript) as scriptable configuration format

Since JSON is based on Javascript at its core (see 4.3.2) and basically just a limited subset of Javascript notation, I use Javascript to construct a configuration loader for JSON (see A.2). I then use that loader to load two configuration files equivalent to the ones for Python, first a fully defined one, then another, shorter, programmatically defined equivalent.

#### 6.2.1 Fully defined configuration

```javascript
// Javascript test configuration example
function config_generator() {
    config = {
        "tests": {
            "numeric test 1": {
                "input": "1+2\n1*2",
                "output": "3\n2",
            },
            "numeric test 2": {
                "input": "2+3\n2*3",
                "output": "5\n6",
            },
            "numeric test 3": {
```
Again clear and verbose, but only really suitable for small sets of individually separate tests. However, unlike the equivalent example for Python, it would not be possible to use linebreaks instead of "\n" here (see 4.3.2). Since Javascript rather than pure JSON is used at this stage, line-comments can also be used at this stage, as seen at the top of the example at 6.2.1.

### 6.2.2 Programmaticallly defined configuration

```javascript
// Javascript test configuration example
function config_generator() {
    config = {
        "tests": {
        
    }
}

// generate some numeric tests
```


count = 0
for (var i = 1; i < 10; ++i) {
    ++count
    test = {
        "input": i + "*" + (i+1) + "\n" + i + "*" + (i+1),
        "output": (i+i+1) + "\n" + (i*(i+1))
    }
    config["tests"]['numeric test ' + i] = test
}

return config
}

Like the Python equivalent, this configuration file is easy to modify to cover a wider range or to include other mathematical operators. On the other hand, it is not quite as readable as the Python equivalent. Most of the difference in this example boils down to the lack of a built-in string-formatting functionality in Javascript, making the definition of the “input” and “output” key-value pairs more awkward both to write and read, in this form.

### 6.3 Conclusion

Though both Python and JSON (through Javascript) are both easily able to fill the role as scriptable configuration format, Python stands out a bit at this stage as being slightly easier to work with. This shows in the way Python provides things such as string formatting and other nice, for a programmer, touches such as the range-generator used in the examples above.

The configuration loaders written for this section demonstrates this even more clearly. The Python configuration loader (see A.1) made use of built-in language features to speed development as a shell-script, and beyond just the configuration loading and generating the configuration loader also contains a few commented-out lines that would allow it to export the generated configuration to YAML or JSON, if desired, all through familiar and commonly-used Python modules.

The Javascript equivalent (see A.2) required more effort to write as no libraries were provided by default, unlike with Python. While there are modules available for Javascript to allow export to, for example, JSON as well, they are not as on-hand as their counterparts in Python, due to Javascript’s lightweight approach[33].
Chapter 7

Embedding Python vs JSON

7.1 Python

Whenever one needs to embed Python in a program, one can always fall back on the reference implementation of Python, CPython[16], written by the inventor of Python in the programming language C. Nearly every other programming language features the ability to bind or in some other way make use of C-libraries, meaning that CPython assures widespread availability of embedded Python.

Beyond the reference implementation of Python there are also several third-party implementations of Python available. One example is Jython[15], a pure Java-implementation of Python only offering Python 2.7 compatibility as of yet. Another is IronPython[14], an implementation of Python on the .NET CLR that also only offers Python 2.7 as of yet. The list of other implementations go on, including implementations in languages such as Javascript and Common Lisp[17].

Looking at the reference implementation of Python in CPython, the tools it makes available are powerful and flexible. Python-code is easily executed, and the results easily examined and data extracted. A good example of this in action is provided in the example of CPython embedded in a C++-program (see B.1), offering the ability to read the example-configurations, both the complete one and the programmatically-shortened one, from the previous chapter. Data can of course be imported into Python-variables with equal ease. For more complex implementations CPython allows the programmer to expose custom C-functions inside the Python scripting-environment as well, allowing for a great deal of flexibility in what the embedded language can do.
7.2 JSON(Javascript)

Javascript, or ECMAScript as it is more formally known, does not have a readily available reference implementation. A reference implementation written in the programming language Standard ML of New Jersey\cite{27} for ECMAScript, fourth edition, is referenced in various sources, but is nowhere to be found, and certainly not for the more current ECMAScript, fifth edition.

Fortunately, since Javascript is immensely popular on the web arena, there is no shortage of third-party implementations available. There is Rhino\cite{34}, for example, the pure Java implementation of runtime-interpreted Javascript managed by the Mozilla Foundation, the developers of the Mozilla Firefox web browser. Another alternative is V8\cite{23}, a JIT-compiled implementation of Javascript written in the programming language C++ by Google, for use in their Google Chrome web browser.

Like the Python reference implementation, the Javascript implementations are also both powerful and flexible. A good example of this is provided in the Java-example with Javascript embedded through Rhino (see B.2), implementing a program to read the example-configurations, both the full and the programmatically condensed one, provided in the previous chapter. In the example data is exported from the Javascript environment, but data can just as easily be imported into the environment. It is also possible to expose Java method-calls to the Javascript environment, allowing the Javascript code to callback into the Java environment. Other implementations of Javascript such as V8 are no less flexible.

7.3 Conclusion

When it comes to available features, Python and Javascript are evenly matched. There are minor differences in the implementations, but nothing that is major enough to impact how easily the two languages are used as embedded languages. Importing and exporting data is a little awkward with both languages, but that is to be expected as type-conversions have to be handled explicitly, as demonstrated in the examples provided for each language. In other respects both languages are also equally comfortable to work with, in embedded form, when comparing CPython and Rhino.

Javascript and Python are equally able to import the configuration formats suggested in the previous chapter, the fully specified configuration as well as those described in shorter form by using programmatically defining part of the configuration, to generate the “real” configuration at run-time.

There is, however, one thing that sets Python apart from Javascript, and that is the ready availability of a fully-functioning reference implementation, written in C so as to be easily interfaced with from most other programming languages. Being able to use the same API to interface with Python from nearly any available programming language is a major benefit. Comparatively, Javascript requires the use of different implementations for embedding Javascript in different
programming languages, or picking one of the third-party implementations such as Google’s V8 in C++, and using that for all programming languages irregardless. While the second alternative may seem equivalent to consistently using CPython, one has to consider that CPython is the official reference implementation of Python and as such unlikely to fall out of use and active development, while using for example V8 leaves one at the whims of Google, who may decide to deviate from the ECMAScript standard, or lose interest in developing it entirely.

Considering that the system I am discussing here is meant to be in use for many years, preferably without significant design changes or continual, major revisions necessary, it would be beneficial to work with a stable, consistent platform that will remain active, and in that regard Python stands out as the winner with CPython.
Chapter 8

Conclusion

Looking back at the questions I set out to answer in section 1.4, let me briefly recap the conclusions reached in this thesis.

1. What principles should this system be designed after, and how do these influence the other choices?

Three key principles were identified. Flexibility (2.1), the proposed system needs to be useful across as broad a range of assignment types as possible, for as many programming languages as possible. Robustness (2.2), as the system would be used in a time-critical situation where a failure would affect potentially hundreds of students. The need for the system to be maintainable with a minimum of effort, and possible to configure without extensive knowledge, also contributes to the need for this principle. Trust (2.3), as the proposed system will be used in a setting where its products will be used to grade students, and it therefore is crucially important that the system can be relied upon to provide results without room for alterations or tampering by the student being graded.

2. What are the security considerations this system needs to be designed to cover?

A section too large to properly summarize, see instead the discussion in section 2.2.2 and 2.3.1.

3. What are the available means to handle those security considerations?

Among the alternatives available to provide security, static analysis of code (3.2.3) is unsuitable due to being easily bypassed in several programming languages, as well as difficult to implement well for multiple different programming languages at the same time. A custom chroot solution (3.2.4) provides a good way to protect sensitive files from access, while at the same time being easy to use, without significant overhead. System call interpositioning (3.2.5) provides another crucial piece to the puzzle, offering a solution to block executables from
accessing restricted system calls, to block access to things such as network sockets or forked processes. System call interpositioning does not impose a heavy system burden in doing so, but requires proper support by the underlying operating system to be properly implemented, which may or may not be possible. Finally there is the virtual machine solution (3.2.6), wherein virtual machines are used to completely isolate the programs being tested from the actual operating system, offering the best possible security, but at a significant performance overhead compared to the previous two alternatives.

4. What configuration format provides the greatest ease of use and flexibility?

Several different, popular configuration formats are compared with an eye towards their readability, their ability to provide nested representations of data and their serializability, or in other words, their ability to be read from and written to disk in several different programming languages. A final, deciding factor is the scriptability of these languages, by which I mean their ability to describe a configuration that can be used to programmatically generate a more extensive set of configurations, a requirement following from the trust-principle mentioned earlier. In this comparison JSON with elements of Javascript (4.3.2) and Python (4.3.6) stand out as good candidates, and after an in-depth comparison in chapter 6 Python comes out as the configuration format offering the greatest ease of use and flexibility for the proposed system.
Appendix A

Configuration loader

A.1 Python

This script was written and tested using Python 3.2.2[19].

#!/usr/bin/env python3

import pprint
import logging
import os.path
import sys
# import yaml
# import json

if __name__ == '__main__':
    if len(sys.argv) != 2:
        print("To start: {0} <config_file>".format(sys.argv[0]))
        exit(1)
    else:
        filename = sys.argv[1]
        if not(os.path.isfile(filename)):
            print("""{0}" not found or not a file!".format(filename))
            exit(2)

logging.basicConfig(level = logging.DEBUG, format="%(asctime)s %(levelname)s %(message)s")
logging.info("Starting!")

generator = __import__(filename[0:-3]).__dict__['config_generator']
logging.info("Config generator loaded!")
logging.debug("Resulting config:
{0}

".format(generator()))

# uncommenting the below lines show the ease of exports to other formats from python
# logging.debug("Resulting config as YAML:
# {0}".format(yaml.dump(generator())))
# logging.debug("Resulting config as JSON:
# {0}".format(json.dumps(generator())))
A.2 Javascript

This script was written and tested using Rhino 1.7.2-r2[34].

```javascript
// (data to be prettyprinted), (levels of indentation),
// (should the first line not be indented)
function prettyPrint(data, indent, skipFirstIndent) {
    var result = ""
    for (var i = 0; i < indent && !skipFirstIndent; ++i) {
        result = result + " "
    }
    if (typeof(data) == "int" || typeof(data) == "float") {
        result = result + "" + data
    } else if (typeof(data) == "string") {
        // substitute actual linebreaks with \n
        result = result + "\" + data.split("\n").join("\n") + "\"
    } else if (data instanceof Array) {
        result = result + "["
        for (var element in array) {
            result = result + prettyPrint(element, indent+1, false) + ",\n"
        }
        for (var i = 0; i < indent; ++i) {
            result = result + " "
        }
        result = result + " ]"
    } else {
        result = result + "{\n"
        for (var key in data) {
            for (var i = 0; i < indent+1; ++i) {
                result = result + " "
            }
            result = result + "\" + key + "\": "
            result = result + prettyPrint(data[key], indent+1, true) + ",\n"
        }
        for (var i = 0; i < indent; ++i) {
            result = result + " "
        }
        result = result + " }"
    }
    return result;
}

//generator = eval(readFile("simple_config.js"))
generator = eval(readFile("complex_config.js"))
print("Config generator loaded!")
config = generator()
print(prettyPrint(config, 0, false))
```
Appendix B

Embedded interpreter

B.1 Python

This program was written with GCC 3.4.6 using the c++98-standard, and linked with Python 3.2.2.

```cpp
#include "Python.h"
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;

// No care is taken to prevent memory leaks as this is a
// short-lived example program.

void printIndent(int indent) {
    cout << setw(indent*2) << "";
}

void print(PyObject* object, int indent, bool initialIndent) {
    if (initialIndent) {
        printIndent(indent);
    }

    if (PyDict_Check(object)) {
        PyObject* keys = PyDict_Keys(object);
        int keyCount = PyList_Size(keys);
        cout << "{" << endl;
        for (int i = 0; i < keyCount; ++i) {
            printIndent(indent+1);
            PyObject* key = PyList_GetItem(keys, i);
            // Further code for printing dictionary keys and values...
        }
    }
    // Further code for handling non-dict objects...
}
```

57
print(key, indent+1, false);

cout << " \";

PyObject* value = PyDict_GetItem(object, key);
print(value, indent+1, false);
cout << "," \" << endl;
}
printIndent(indent);
cout << "}" \" << endl;
}
else if (PyList_Check(object)) {
  int keyCount = PyList_Size(object);

  cout << "[" \" << endl;
  for (int i = 0; i < keyCount; ++i) {
    printIndent(indent+1);
    PyObject* value = PyList_GetItem(object, i);
    print(value, indent+1, false);
    cout << ",";
  }
  printIndent(indent);
cout << "]" \" << endl;
}
else if (PyUnicode_Check(object)) {
  int dataSize = 256;
  wchar_t* data = PyUnicode_AsWideCharString(object, &dataSize);
  char* temp = new char[dataSize+1];
  int copied = wcstombs(temp, data, dataSize);
temp[dataSize] = '\0';
cout << temp;
}
else {
  PyObject* type = PyObject_Str(PyObject_Type(object));
  int dataSize = 256;
  wchar_t* data = PyUnicode_AsWideCharString(type, &dataSize);
  char* temp = new char[dataSize+1];
  int copied = wcstombs(temp, data, dataSize);
cerr << copied << " == " << dataSize << endl;
temp[dataSize] = '\0';
cout << (Unhandled type: " << temp << ");
}

int main(int argc, char* argv[]) {
  // convert path from char* to wchar_t*
  int pathSize = strlen(argv[0]);
  wchar_t* path = new wchar_t[pathSize+1];
  mbstowcs(path, argv[0], pathSize);
APPENDIX B. EMBEDDED INTERPRETER

```c
Py_SetProgramName(path); // set path

// init python
Py_Initialize();

// add local directory to path
PyRun_SimpleString("import sys; sys.path.append(".");");

// import parse module
char* parseModule = "custom_parser";
PyObject* moduleName = PyUnicode_FromString(parseModule);
PyObject* module = PyImport_Import(moduleName);
if (module == NULL) {
    cerr << "failed to find module: " << parseModule << endl;
    exit(1);
}

// get function from module
char* functionNameChar = "read_config";
PyObject* functionName = PyObject_GetAttrString(module, functionNameChar);
if (!PyCallable_Check(functionName)) {
    cerr << "cannot call function: " << functionNameChar << endl;
    exit(2);
}

// call the function
PyObject* functionParameters = PyTuple_New(1);
PyObject* parameter = PyUnicode_FromString("sample_config.py");
PyTuple_SetItem(functionParameters, 0, parameter);
PyObject* functionResult = PyObject_CallObject(functionName, functionParameters);
if (functionResult == NULL) {
    cerr << "Call failed:" << endl;
    PyErr_Print();
    exit(3);
}

// check the result
if (!PyDict_Check(functionResult)) {
    cerr << "Result was not a dictionary!" << endl;
    exit(4);
}

print(functionResult, 0, true);

Py_Finalize(); // release associated resources
return 0;
```
B.2 Javascript

This program was written with Java 1.7.0.05 using Rhino 1.7R4.

```java
import java.io.File;
import java.io.IOException;
import java.util.Scanner;
import java.util.Set;
import org.mozilla.javascript.ConsString;
import org.mozilla.javascript.Context;
import org.mozilla.javascript.NativeArray;
import org.mozilla.javascript.NativeObject;
import org.mozilla.javascript.Scriptable;

public class JavascriptEmbedded {
    private final static String linebreak = System.getProperty("line.separator");

    private static String getIndent(int indent) {
        StringBuffer result = new StringBuffer();
        for (int i = 0; i < indent; ++i) {
            result.append(" ");
        }
        return result.toString();
    }

    private static String print(Object data, int indent, boolean initialIndent) {
        String result = "";
        if (initialIndent) {
            result += getIndent(indent);
        }

        if (data instanceof Double) {
            result += String.format("1.2f", (Double)data);
        }
        else if (data instanceof String || data instanceof ConsString) {
            // substitute actual linebreaks with \n
            String[] lines = data.toString().trim().split("\n");
            boolean first = true;
            result += "\n";
            for (String line : lines) {
                if (!first) {
                    result += linebreak;
                }
                first = false;
                result += line;
            }
            result += "\n";
        }
        return result;
    }

    public static void main(String[] args) {
        JavascriptEmbedded embed = new JavascriptEmbedded();
        String result = embed.print("Test string", 10, true);
        System.out.println(result);
    }
}
```

else if (data instanceof NativeArray) {
    result += "[" + linebreak;
    NativeArray array = (NativeArray)data;
    for (int i : array.getIndexIds()) {
        Object element = array.get(i);
        result += print(element, indent+1, true) + "," + linebreak;
    }
    result += getIndent(indent);
    result += "]";
}
else {
    result += "{" + linebreak;
    NativeObject mapping = (NativeObject)data;
    Set<Object> keys = mapping.keySet();
    for (Object key : keys) {
        result += print(key, indent+1, true) + ": ";
        result += print(mapping.get(key), indent + 1, false) + "," + linebreak;
    }
    result += getIndent(indent);
    result += "}";
}
return result;

public static void main(String[] args) throws IOException {
    // create and init javascript
    Context jsContext = Context.enter();
    try {
        Scriptable jsScope = jsContext.initStandardObjects();

        // read config
        String configFilename = "/complex_config.js";
        String config;
        {
            StringBuffer buffer = new StringBuffer();
            Scanner scanner = new Scanner(new File(configFilename));
            try {
                while (scanner.hasNext()) {
                    buffer.append(scanner.nextLine()).append(linebreak);
                }
            } finally {
                scanner.close();
            }
            config = buffer.toString();
        }

        // define the generator function
        jsContext.evaluateString(jsScope, String.format("generator = %s", config),
"functionDefine", 0, null);

// call the generator function
NativeObject result = (NativeObject)jsContext.evaluateString(jsScope,
    String.format("generator()", config), "functionCall", 0, null);

// parse the data from the generator function
System.out.println(print(result, 0, true));

} finally {
    // dispose of javascript resources
    Context.exit();
}
}
Bibliography


[31] Roland McGrath and Free Software Foundation. *Chroot - run command or interactive shell with special root directory*.


