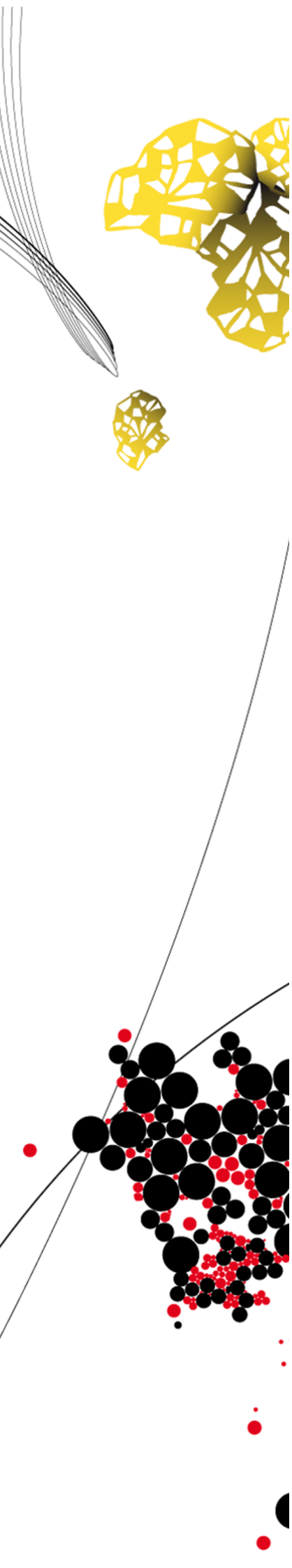

UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

Java Code Virtualization of Industrial-strength Java Code

Gert Jan Laverman

M.Sc. Thesis

July 2016

Supervisors:
prof. dr. M. Huisman
dr. M. H. Evers
ing. A. Huinink

Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

THALES

Approval Internship report/Thesis of: ***Gert Jan Laverman***

Title: ***Java Code Virtualization of Industrial-strength Java Code***

Educational institution: ***University of Twente***

Internship/Graduation period: ***October 2015 – June 2016***

Location/Department: ***Thales Hengelo, Interface Products***

Thales Supervisor: ***Arnold Huinink***

THALES

This report (both the paper and electronic version) has been read and commented on by the supervisor of Thales Netherlands B.V. In doing so, the supervisor has reviewed the contents and considering their sensitivity, also information included therein such as floor plans, technical specifications, commercial confidential information and organizational charts that contain names. Based on this, the supervisor has decided the following:

- ✓ This report and/or a summary thereof is **publicly available to a limited extent (Thales Group Internal)**.
  It will be read and reviewed exclusively by teachers and if necessary by members of the examination board or review committee. The content will be kept confidential and not disseminated through publication or inclusion in public libraries and/or knowledge bases. Digital files are deleted from personal IT resources immediately following graduation, unless the student has obtained explicit permission to keep these files (in part or in full). Any defence of the thesis may take place **in public to a limited extent**. Only relatives to the first degree and teachers of the ***Faculty of Electrical Engineering, Mathematics and Computer Science*** department may be present at the defence.

Thales Nederland B.V. and University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science have agreed that the report will be kept confidential and will not be included in a public library or knowledge base until July 31, 2021.

Thales Nederland B.V. and University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science have furthermore agreed that the candidate may defend his work in public, adapted and anonymized in such a manner as to not disclose any confidential information related to the project.

Approved:

(Thales Supervisor)

Approved:

(Educational institution)

Hengelo 18 April 2016

(city/date)

University of Twente

M.Sc. Thesis

Performed at Thales Netherlands

Java Code Virtualization of Industrial-strength Java Code

Author:
G. J. Laverman

Supervisor:
prof. dr. M. Huisman

Co-assessor:
dr. M. H. Everts

Supervisor Thales:
ing. A. Huinink

July 15, 2016

Abstract

Background Java is a popular object-oriented general purpose computer programming language that uses an intermediate bytecode format representation of a program to be interpreted by the Java Virtual Machine. The architecture-neutral intermediate bytecode design principle is however more susceptible to reverse engineering than computer programs written in a language that compiles source code to machine-specific object code. Additional security measures are necessary if a program's bytecode contains sensitive code such as intellectual property or trade secrets that must be kept secret. Protecting Java bytecode against reverse engineering attacks is however no trivial task. There are some techniques known such as code obfuscation or code offloading, but the former is not sufficient to stop determined attackers and the latter is not applicable to systems that have to operate standalone in a closed environment. Code virtualization is a technology that could possibly improve the resilience of Java bytecode against reverse engineering attempts. Using code virtualization as a technology to protect Java bytecode from reverse engineering is however relatively new and not much is known yet about its effectiveness and real life performance. This report investigates these unknowns by applying code virtualization to sorting algorithms and a demo application. The sorting algorithms have different space and time complexity classes used to investigate compatibility and the scalability of the virtualization technology while the demo application reflects a more realistic use case with multiple components working together.

Results Benchmarks measuring the performance of sorting algorithms and their encrypted and virtualized counterparts show that there is a performance penalty for applying additional protection to a Java program. The performance runtime of an encrypted version of the reference sorting algorithms runs a factor 1 to 1,5 slower depending on the algorithm. This is minimal overhead but the offered protection is not sufficient against determined attackers. Code virtualization offers arguably stronger protection over existing obfuscation techniques and requires a lot more effort to reverse engineer. The protection/performance trade-off is however significant. For virtualized versions of the sorting algorithms the runtimes increased with a factor 100 on average. The protection/performance trade-off can be tweaked by adjusting parameters but the performance penalty remains significant with minimal settings. The knowledge and experience from these experiments have been used to develop the demo application, which reflects a more realistic use case, to determine if virtualization can be applied at a reasonable cost.

Conclusions Applying the advanced code virtualization protection technology to a program enhances its protection against reverse engineering. Performance however deteriorates significantly for the virtualized program code. Protecting code in real-time applications requires therefore careful consideration and preferably a thorough comparative protection/performance trade-off assessment.

Preface

The business line Above Water Systems (AWS) of Thales Netherlands develops high-tech combat management systems and integrates other naval systems and services for surface vessels. Thales TACTICOS is an open system architecture, surface ship Command & Control System that integrates hardware from various suppliers worldwide. Most of TACTICOS has been written in Java, which is susceptible to reverse engineering. Thales wants to protect its intellectual property from theft when the software is operated off the premises during development at a subcontractor or when operated in a production environment on a naval vessel.

This thesis describes the results of a final graduation project performed at Thales Netherlands to investigate the possibilities of Java code-virtualization to protect TACTICOS. It has been submitted in partial fulfillment of the requirements of a Master of Science degree at the University of Twente. The thesis contains a background study on Java code virtualization and related topics such as code obfuscation, bytecode decompilation and reverse engineering. A design for a demo application called TACTLESS, that represents the TACTICOS technology stack without containing any intellectual property from its source code, is presented and evaluated. The thesis concludes with results gathered from benchmarks and tests performed on sorting algorithms and the TACTLESS demo application.

Due to the sensitive nature of the research it has been declared confidential. The full report is therefore not publicly available. Thales Nederland B.V. and University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science have agreed that the report will be kept confidential and will not be included in a public library or knowledge base until July 31, 2021

This document has been written under supervision of prof. dr. M. Huisman, professor in Formal Methods and Tools, faculty of Electrical Engineering, Mathematics and Computer Science at the University of Twente and ing. A. Huinink, Software Engineering expert at Thales Netherlands.

Gert Jan Laverman
November 28th, 2015

Acknowledgement

I would like to express my gratitude to advisor prof. dr. M. Huisman for clearing time in her busy schedule and guiding me during the preparation for my graduation project.

Besides my advisor I would like to thank ing. A. Huinink, my guide and coach at Thales Hengelo, for sharing his extensive knowledge on TACTICOS and software development expertise in general with me. He introduced me to the different teams that I came in contact with during the analysis of the TACTICOS technology stack and has provided encouragement and guidance during my stay there ever since. I would also like to thank him for proof reading this report and acknowledge that his insights greatly assisted me during my research.

My sincere thanks also goes out to M. Beekveld for formulating the assignment, inviting me for an introductory interview and granting me the privilege of carrying out this assignment for my graduation project. I would also like to thank him for staying involved in the background and in particular for pulling strings to make it possible to experiment with Solidshield.

A special thanks goes out to IT security expert B. Marcon from the ICT Security Unit at ThereSIS Innovation lab, for providing the service to apply Solidshield code obfuscation and virtualization to our Java programs and for sharing his knowledge on Solidshield.

Most importantly, I would like to emphasize that none of this would have been possible without the love and support of my family.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Thales Netherlands is a Dutch branch of the international operating Thales Group. With in-depth knowledge of platforms, weapons, sensors, communication, electronic warfare, navigation and other mission system components as well as the management engineering and integration skills they can successfully define and realize complete mission system solutions. Thales is a supplier and integrator of complete missions system solutions for surface ships and acts as the lead system integrator on behalf of their clients.

The highly successful TACTICOS [25] Combat Management System (CMS) captures diverse user requirements. It builds on a continuous evolution in hardware, middleware, software and operational applications to deliver a fully distributed system architecture for tactical picture compilation, decision support, unit and force coordination, sensor and weapon assignment, information exchange, mission planning and embedded training. TACTICOS is suitable to operate on naval vessels of all sizes and for all missions. It is the world's most successful CMS used by over twenty leading navies world-wide [25].

Nations nowadays often negotiate industrial participation for their national industries when spending tax money on defense projects. Thales must agree to buy products or services from the client nation in these defense offset agreements as a compensation for the placed order. When outsourcing production or development to third parties is part of the industrial compensation agreement then special care has to be given to protect trade secrets and unique selling points. A nation's intelligence service or subcontractor might intentionally try to steal confidential information like trade secrets to use in their own defense programs or to seek for vulnerabilities in the system that could be exploited against hostile users. Then there is also the risk of subcontractors leaking information unintentionally due to bad practices or lack of security.

Thales wants to protect its intellectual property from theft and prevent it from falling into the wrong hands when collaborating with subcontractors outside the Thales premises.

Thales deploys a number of techniques to protect its code against reverse engineering. Each of these has advantages and disadvantages. This report will investigate the applicability of code virtualization technology on the Java based TACTICOS Command & Control System.

The focus lies hereby on the Solidshield protection technology that incorporates obfuscation and virtualization strategies to increase the effort required to

reverse engineer an application.

1.1 Motivation

Java source code compiled to hardware and operating system independent binary Java bytecode contains instructions for the Java Virtual Machine (JVM). Most of the original source code information remains however available in the Java bytecode. Therefore the bytecode is relatively easy to reverse engineer by decompilation [7].

There are several approaches to counter reverse engineering attacks, such as code encryption, code obfuscation, offloading, etc. but none have been effective enough so far.

Thales wants to investigate if Java code virtualization can protect its leading-edge Java-based products against reverse engineering and evaluate how it will affect the performance and stability.

Solidshield is a new tool by Tages, a former co-venturer specialized in binary code protection against code tampering, reverse-engineering and piracy, that promises strong protection with a marginal overhead off ten to thirty percent. The Solidshield tool can protect Java Archive (JAR) files by encrypting, obfuscating and virtualizing the bytecode inside the JAR archive. On paper this offers good prospects for enhanced protection with a relatively small performance penalty. Thales is therefore especially interested in this technology and wants to evaluate the maturity of the Solidshield technology to determine if it is a viable solution for their main concern regarding reverse engineering of their software.

In order to validate the effectiveness of Java code virtualization from a security perspective the TACTLESS demo application prototype has been developed which contains the technology stack of TACTICOS without containing the actual intellectual property. This TACTLESS demo application will represent the CMS and can be shared with third parties outside Thales premises without risking exposure of intellectual property. It can be handed out to security experts to test the strength of the protection or to suppliers and subcontractors in order to solve problems that originate from the virtualization technology without exposing the actual TACTICOS code base.

The TACTLESS demo application will also act as a testing environment to run benchmarks and tests on a smaller and nimbler code base without the complexity of the full TACTICOS system with its interfaces. Sorting algorithms are introduced as an important component in these tests and included in TACTLESS. The first step is to determine if the code virtualization is applicable to the current code base without breaking functionality or external libraries used in the technology stack. Subsequently we will move on to benchmark the virtualized sorting algorithms and TACTLESS demo application's performance in terms of memory complexity and time complexity. The results are compared to the unprotected code and we validate the virtualized code to ensure the behavior is (observable) equivalent.

Based on the results a recommendation has been formulated to aid in the decision whether Thales should adopt the Solidshield technology for their Java-based systems.

1.2 Research Question

Thales is interested to find a protection technology that can protect their Java code against reverse engineering. To capture that goal the following research question is proposed:

> **Research question** *How can code virtualization and code obfuscation provided by Solidshield contribute to the protection against reverse engineering of intellectual property in the Thales Java based technology stack used in their Combat Systems?*

Protecting Java bytecode from reverse engineering is a very interesting problem. The portability of Java across platforms has made Java one of the most popular programming languages used around the world. This portability is achieved by the intermediary Java bytecode format that is run inside a JVM. The portability and versatility has however also drawbacks. One major issue is the fact that Java's intermediary bytecode format can easily be reverse engineered by decompiling it back to readable source code. When a company puts effort into developing software they do not want to loose their intellectual property to competitors or other parties with the ability to reverse engineer the source code from the Java bytecode for a fraction of the development costs. With the source code they can alter the product, steal trade secrets, remove copy protection mechanisms, etc. Because Java is widely used for many applications in very diverse fields there is a demand for protecting Java bytecode against reverse engineering attacks. Especially for highly sophisticated systems with components that require confidentiality, such as the defense and security systems that Thales provides this is a major issue. Therefore the research question is very interesting with broad applicability beyond the protection of intellectual property captured in Java bytecode.

In order to answer the main research question the problem has been divided into sub-problems that have been defined in the following sub-questions:

Sub-question 1.1 *Is Solidshield code virtualization compatible with the TACTICOS technology stack?*

Research question 1.1 is an important one because TACTICOS is a sophisticated application using several advanced language features such as Java introspection and reflection that might not be supported by the Solidshield virtualization technology. The TACTLESS demo application that has been developed contains most of the techniques, patterns, libraries, etc. used by the TACTICOS technology stack. This has been used to verify if it remains functionally equivalent after virtualization and to identify possible obstacles. Discovered restrictions imposed by Solidshield might be acceptable if they require minor code changes to overcome. Incompatibility with important mechanisms implemented by TACTICOS or third party software in the technology stack will however be problematic.

Sub-question 1.2 *How should the TACTLESS demo application look like?*

Research question 1.2 is a followup question on the previous question. It is important to properly design the TACTLESS demo application because it must act as a independent research application that is representative for the technology and language features used in TACTICOS. Therefore the key technology features in the TACTICOS technology stack have been identified. The design based on the analysis is implemented in a novel way without any resemblance to the original code.

Sub-question 1.3 *How does Solidshield code virtualization impact TACTICOS in terms of performance?*

Answering research question 1.3 is vital for the formulation of a well-motivated recommendation. Without performance figures it is impossible to determine if the protection/performance trade-off is acceptable. Appropriate metrics have been identified to express and evaluate the performance figures. These metrics have been extracted from sorting algorithms and the TACTLESS demo application. Benchmarking must give insight in how much the Solidshield protection affects the runtime behavior of a protected program compared to the original program.

Sub-question 1.4 *How does Solidshield code virtualization affect the behavior of TACTICOS in terms of reliability and stability?*

Research question 1.4 asks another important sub-question because reliability and stability are also very important quality aspects to consider. Testing techniques must be applied to investigate if Solidshield affects reliability and robustness of the application. Solidshield claims to maintain functional equivalency but their code obfuscation and virtualization techniques alter the control flow and might introduce concurrency issues not present in the original program.

Sub-question 1.5 *How effective is code virtualization to prevent reverse engineering of the code?*

Research question 1.5 is very important. The offered protection must be very good to justify the hassle involved in implementing and applying it during the software development and maintenance process. By experimenting with code virtualization and using theory from the literature a confidence indication can be formed regarding the resilienceness of code virtualization to reverse engineering.

Sub-question 1.6 *How secure is the Solidshield protection regarding the prevention of reverse engineering?*

Research question 1.6 reflects one of the biggest questions. By reasoning based on theory provided in related literature it is possible to derive an estimation backed by arguments at best. The actual evaluation of the resilience of the Solidshield protection against reverse engineering attacks is however beyond the scope of our assignment due to restrictions in time and resources. This is best left to specialists in the field of computer security. External security experts could therefore analyze the project deliverables and attempt to crack the TACTLESS demo application but that will be considered as future work.

1.3 Assumptions

The following assumptions have been made as an initial starting point for this research project:

Assumption 1 *Code virtualization combined with code obfuscation makes it very hard to reverse engineer intellectual property from the bytecode.*

A_1 *explanation:* It is virtually impossible to guarantee total protection against reverse engineering of Java applications. The combination of code virtualization with code obfuscation gives arguably better protection then other well-known technologies explored in the literature so far.

Assumption 2 *Code virtualization and code obfuscation will affect the performance of the Java code.*

A_2 *explanation:* Virtualized code run in a virtual machine adds additional overhead compared to native code. Code obfuscation tactics typically also add overhead. Therefore performance penalties are very likely to occur. This might have consequences for time sensitive aspects, i.e. delays in real-time data or critical timing issues in the message transport layer, etc.

Assumption 3 *The Solidshield implementation affects system behavior potentially losing functional equivalency.*

A_3 *explanation:* When certain code obfuscation tactics are applied the code and control flow get altered. This might have consequences for runtime facilities in Java such as reflection, introspection, meta-data, etc. which in turn might have consequences for concurrency aspects leading to unintended behavior due to race conditions, starvation, deadlocks, etc.

1.4 Approach

This thesis documents the research, design and evaluation phases of the graduation project. The research phase contains a background study on the applicability of the Solidshield protection technology to protect Java programs against reverse engineering. It has been performed in preparation for the design and evaluation phases to gain insight in the topic related theory and technology. The design phase addresses the design and development of the TACTLESS demo application that represents the TACTICOS technology stack to be used during the evaluation phase. A short description of these three phases is given below.

Research

During the research phase an extensive literature study has been performed to investigate which technologies are available and to explore related work. Special attention has been given to the Solidshield technology as Thales has expressed the desire to evaluate this technology with their products. There is not much known about this product therefore we had to experiment with the technology

ourselves to learn what we needed to know.

Research question 1.1 is mainly a practical one. Based on the referenced literature on obfuscation and virtualization possible problem areas regarding the Java programming language have been identified. This knowledge has also been used to analyze the potential problem areas in the TACTICOS technology stack. The suspected problems have been simulated in a demo application to verify if the obfuscation and virtualization works. This TACTLESS demo application contains most of the techniques, patterns, libraries, etc. used by the TACTICOS technology stack and is used to verify if it remains functionally equivalent after applying protection to its code. Discovered restrictions imposed by Solidshield might be acceptable if they require minor code changes to overcome. Incompatibility with important mechanisms implemented by TACTICOS or third party software in the technology stack are however problematic.

Prior to the TACTLESS demo application a small sorting program has been developed with a few basic elements during the research phase. The experiences and knowledge gained with the sorting program have later been used to design TACTLESS in the design phase.

For research question 1.2 an extensive analysis of the TACTICOS technology stack has been performed. A selection of important components has been made from the involved techniques, patterns, libraries and other relevant involved technologies. These components have been analyzed and assessed on their susceptibility for potential problematic behavior after being obfuscated and virtualized. This information has been used to come up with a design including the important technology to represent the TACTICOS system and allowed proper testing and evaluation of the obfuscated and virtualized code in a representative manner. Based on the requirements of Thales and the analysis of the current technology stack a priority analysis has been performed to determine the implementation order of features during the iterative development process.

During the research phase the analysis has been limited to general language features provided by Java such as threading, introspection and reflection. This has been done to limit the initial scope to core language features that we felt were important. The motivation behind this decision was that given the limited time this would be indicative for the virtualization technology as a whole. When these language mechanisms were properly supported then it would be fair to expect that it would imply that any subset implemented in Java should also be compatible. The extensive analysis of the whole TACTICOS technology stack has been performed during the design and implementation phase and the resulting TACTLESS demo application has been subjected to benchmarks and tests in the evaluation phase as explained later.

In order to answer 1.3 we benchmarked the sorting algorithms and the TACTLESS demo application. Based on literature we selected several metrics and devised a benchmarking approach tailored to our needs. The original sorting algorithm measurements were compared to different parameterized obfuscated and virtualized versions. This provided a solid basis for reasoning regarding the effects of virtualization later on.

During the research phase we focused on time complexity by measuring

runtimes of the chosen algorithms. Although they already gave an indication regarding performance they were insufficient to explain some of the observed behavior during the first tests. Based on this experience the improved TACTLESS demo application has been developed. The intensive testing with TACTLESS took place during the evaluation phase. Runtime complexity, memory complexity and Java virtual machine metrics have been collected during these tests.

It is challenging to give a definitive answer to research question 1.4. As expressed in assumption 3 the bytecode is altered by code obfuscation and virtualization potentially introducing unexpected behavior not present in the original program. Although Tages claims that their Solidshield tooling produces functionally equivalent code they have not formally verified their code. This means that even though there are known obfuscation tactics that transform code into functionally equivalent code it remains unclear whether these transformations have been implemented correctly in Solidshield. Therefore Thales has to validate the virtualized versions of their software to ensure that it still behaves correctly. This validation is done by simulating with the demo application based on the specification constructed during the design and implementation phase. Formal verification is difficult because the virualization acts as a big black box protecting the virtualized code from prying eyes. This also means that the use of static analysis will be limited. Therefore dynamic analysis techniques during simulations have been attempted. Testing has been included in the iterative development process to discover faults and problems during the implementation phase. This allowed us to provide feedback to Tages early on and cooperate with them to improve their tooling where necessary.

For research question 1.5 we rely on theory from the literature. By referring to results from published papers an indication is given regarding how promising the code virtualization technique is to prevent the reverse engineering of bytecode.

Research question **??** is relatively easy to answer. If research question 1.1 has been answered positively then we can extend that by adding external obfuscation to the development chain. Thales currently uses a tool that obfuscates their software. It is possible to determine if this tool is capable to work with the altered bytecode produced by Solidshield and vice versa. This is done by applying Solidshield to the extra obfuscated version and by applying extra obfuscation to the Solidshield version. These versions can be subjected to our testing approach. If they are compatible the sequence for applying these tools can be determined for the best result.

For research question 1.6 the concept of threat levels are introduced. Based on theory from the literature and experiments done during the Evaluation phase an indication is given to indicate how well the protection works to the defined threat levels. For a full security analysis however we advise to send an obfuscated and virtualized version of our demo application to an external software security firm. They poses the knowledge, expertise and resources rivaling the highest threat levels and can take the testing beyond the scope of this project.

The research phase has been concluded with a small sorting program con-

taining several sorting algorithms from different complexity classes to get a first impression of the performance of the virtualized code compared to the original code. The sorting program is discussed in detail in Chapter 4.

Design and Implementation

During the design phase an outline has been created for the TACTLESS demo application that is representative for the relevant aspects and technology stack used in the TACTICOS CMS. The TACTLESS demo application is used to evaluate Solidshield in a realistic setting without any traces of code or intellectual property present in TACTICOS and accompanying technology stack. It has to contain therefore all the technology aspects discovered during the research phase but implemented in a novel way without any relations or resemblance to the original code base. This clean room design requirement ensures that there is no intellectual property exposed from the TACTICOS CMS when the developed demo application is subjected to reverse engineering attacks attempts by security experts off the premises.

The TACTLESS demo application also serves as a vehicle for future communication regarding Solidshield or alternative protection technologies by providing a way to debug and request support on directly related issues that might arise and require sharing of code examples with (untrusted) external third parties to recreate the faults. It has a modular design that can be extended or adapted. Functionality and components can be easily added or altered on a later moment. The sorting algorithms have been converted to such components and added to TACTLESS for testing purposes.

Evaluation

The evaluation of the protection technology primarily entails determining if the virtualized code behaves functionally equivalent to the original demo application code. Part of the evaluation is comparing the code of the regular unprotected vanilla code to the virtualized version. Measurements have been taken during benchmarking tests to determine the consequences for memory-usage, CPU performance, latencies and timing on runtime. The results have been used to assess the impact of Solidshield protection technology on the entire product life cycle, ranging from software design, software development and product support.

1.5 Structure of the Report

The remainder of this report is structured as follows:

Chapter 2 provides a background context and reflects our findings on relevant related work encountered during a literature study on the subjects of reverse engineering, decompilation, bytecode encryption, code obfuscation & control flow obfuscation, code virtualization and the evaluation of Java programs. A few encryption and obfuscation solutions are named and related to the threat levels. Finally the chapter concludes with a recapitulation discussing the concepts extracted from the literature and the derived insights after studying the varying topics.

The methodology for testing and evaluating the programs created during this research is discussed in Chapter 3. The benchmarking approach is explained in detail and the choice of metrics that have been collected are motivated.

In Chapter 4 there are several sorting algorithms introduced and a sorting program that implements these algorithms. This sorting program is used for testing and evaluating the effects of code obfuscation and code virtualization. The choice of algorithms for the sorting program are mentioned and motivated in Chapter 4.1, 4.2 and 4.3.

The tool that is used to obfuscate and virtualize the Java bytecode is introduced and examined in Chapter 5. Protected code is analyzed and the working of the protection tool is derived from these protected programs.

Chapter 6 describes some components from the technology stack for TACTICOS. Here the components are discussed that need to be implemented in the TACTLESS demo application.

The TACTLESS demo application created as part of this research is discussed in Chapter 7. First the important Open Services Gateway Initiative (OSGi) concepts are mentioned in Chapter 7.1 and Chapter 7.2 before presenting the design in Chapter 7.3.

The research results are presented and discussed in Chapter 8. In Chapter 8.1 the focus lies on the results gathered from benchmarking and evaluating the sorting algorithms. Chapter 8.2 shows the migration from the sorting algorithms to sorting bundles in an OSGi environment and the results from applying code virtualization to the TACTLESS demo application.

Chapter 10 is dedicated to summarize the findings discussed in this report and divided into the following sections. Chapter 10.1 contains a summary of the thesis. Contributions and limitations of the current work are discussed in Chapter 10.2. Limitations of the current work and recommendations for improvement are given in Chapter 10.3 and Chapter 10.4. Future work is proposed in Chapter 10.5. Finally the conclusions and recommendations are presented in Chapter 10.6.

Chapter 2

Background

The process of software engineering nowadays typically includes writing source code in a programming language and transforming the high level source code to a lower level machine code that can be executed directly by a machine or an intermediary language. The former is achieved by compiling the source code with a compiler into machine code also known as object code while the latter typically relies on an interpreter such as a virtual machine. Object code is specific for the chosen Central Processing Unit (CPU) architecture and can be directly executed by a compatible processor. When the application has to be run on a different architecture then the source code hast to be re-compiled from scratch to generate the object code compatible with the CPU instructionset for that architecture.

Java is a popular high-level programming language that provides platform and operating system independence trough the use of an intermediary bytecode format run on a virtual machine. The bytecode is comparable with object code generated by a compiler but instead of platform specific object code executed directly by a CPU the Java bytecode is run on a JVM that translates the bytecode instructions indirectly to object code for the underlying CPU. This would theoretically allow a Java program to run on any compatible platform that has a JVM available without requiring alterations to the object code. The distribution of Java bytecode make it easier to decompile Java applications back to readable source code compared to object code because it contains higher level information required by the virtual machine to interpret and translate to the underlying machine architecture.

A literature study has been performed to gain knowledge about reverse engineering of Java bytecode and the documented technologies that aim to prevent this from happening. Performance and space/time complexity were expected to be affected, therefore we looked into performance evaluation for Java to properly evaluate the performance and possible unforeseen side effects.

2.1 Reverse Engineering

There are many definitions and examples of reverse engineering documented. Some definitions are very broad and cover a large domain of different engineering disciplines and others give more specific definitions applicable to software

engineering. They range from picturing it as the act of extracting knowledge or design blueprints from anything made by man [9] to more specific descriptions where it is considered a process of analyzing a system to identify the components and their relationships to recreate a representation of that system on a higher level of abstraction [6].

The traditional software engineering development process typically starts by creating a model with concepts describing the system to be build and adding lower-level implementation details along the way towards a low level concrete system implementation i.e. starting with a Unified Modeling Language (UML) model, implementing it in a programming language and then compiling the source code into an executable binary.

This 'normal' software development trajectory is sometimes referred to as forward engineering [6] in contrast to reverse engineering. If forward engineering involves designing models and writing source code during the implementation phase to create a program then reverse engineering could be regarded as the opposite. Reverse engineering a program or application would generally involve obtaining the object code or bytecode and recover readable source code or models that are functionally equivalent to the original artifact. This is sometimes called reverse code engineering [28].

From here on whenever we use the term reverse engineering we actually refer to the practice of reverse code engineering unless otherwise specified.

Software reverse engineering can be divided in two categories as explained by [9]. The security related category entails malicious software, reversing cryptographic algorithms, Digital Rights Management (DRM), auditing program binaries, etc. The second category applies to reverse software development where the goal is not to produce a new program from scratch but instead use a concrete system as starting point and work from there. This can be done to achieve interoperability with proprietary software, develop competing software, evaluating software quality and robustness, etc.

Reverse engineering is in principle a neutral activity, just like forward engineering it can be used to develop software in a legitimate way when it is applied to own work. It can be used in parallel to regular forward engineering for round-trip engineering [18], higher level abstraction design recovery such as generating models from code [4], etc.

When the process is applied to copyrighted material the dividing line between legal and illegal becomes fuzzy. It can be used to crack DRM protection schemes for example or to take apart a competitors application and steal intellectual property or trade secrets from the program code.

In his book on intellectual property and open source Lindenberg dedicates one chapter on reverse engineering. The focus of his work lies on the juridical aspects of protecting code such as patents, copyright, trademarks, trade secrets, contracts, licensing, etc. [16].

Based on reverse engineering jurisprudence from the past, several examples are given as guideline for acceptable reverse engineering that holds up in court. The important lesson here is that an author can not rely on copyright and legislation to prevent reverse engineering from happening to his work. Reverse

engineering is a common practice and has been for quite some time. In fact software is easier to reverse engineer than traditional analogue systems because once the essence of the program logic has been recovered it is relatively easy to re-implement and duplicate at a greatly reduced cost compared to the investment of the original creator who had to put effort and time into creating and implementing the original design.

Therefore developers need to be aware of the consequences of distributing their intellectual property and trade secrets coded inside programs. As said reverse engineering is not necessarily a good or bad practice. It depends on the context of how it has been applied and with what intend but also on the perspective from the observer. To illustrate this I would like to include the following example:

One well-known case of reverse engineering in the relatively young information technology industry was the cloning of the IBM PC by Compaq in the early 1980's. The IBM PC had an open architecture and was build from components already available on the market to keep costs low. Only the BIOS chip was a proprietary design and produced by IBM. Compaq could therefore get all the components from IBM's suppliers or other competing vendors except the proprietary BIOS chip. To build their own IBM compatible PC clone they had to reverse engineer the IBM BIOS chip. Their efforts to clone the IBM BIOS chip spiked the personal computer revolution and it only took them fifteen senior programmers, one million dollars and several months to accomplish [16].

This anecdotal evidence is just one of the many examples showing that reverse engineering to recover intellectual property happens. It shows that successful attempts can be accomplished at a fraction of the development costs while yielding great results regarding the return value if successful.

Obviously IBM was not amused to loose money to the copycats from Compaq and other computer manufactures that followed their example. However, from the general public's perspective the reverse engineering of the IBM BIOS chip was a positive one because it resulted in a revolution in personal computing that otherwise would have been postponed or might never have taken place.

I took the liberty of including the Compaq example because besides being a good and well-known example with huge impact on the entire industry it also affected me personally. My first IBM compatible personal computer was made by Compaq. They changed the world and they shaped my future. Without affordable yet capable computers in my youth I might have followed a different career path.

2.2 Decompilation

Java bytecode is relatively easy to decompile back into Java source code with decompilers. In a way it is only a matter of reversing the compilation strategy [20]. Java bytecode is by nature more susceptible to reverse engineering than compiled machine code because it contains a higher-level representation of a program. The symbolic information inside the bytecode such as complete type signatures, method invocations, etc. are necessary for dynamic linking and loading and make Java much more prone to decompilation [21].

Decompilers such as Fernflower [27] and Proycon [24] take advantage of these

core principles of the Java programming language. They can produce readable source code from Java bytecode that will look almost identical to the original source code used to compile the bytecode. These automated tools make reverse engineering Java programs relatively effortless and without additional measures there are virtually no barriers for one to apply these tools to bytecode of a Java program and recover usable source code.

Any Java program can be reverse engineered by a competent and determined programmer given enough time and effort. There are several strategies to protect code against decompilation varying from creative ideas such as selling the source code to more realistic techniques like DRM, using native methods via Java Native Interface (JNI) or Java Native Access (JNA), server side execution, encryption (discussed in Chapter 2.3), obfuscation (discussed in Chapter 2.4), etc.

The idea of selling source code for an additional yet reasonable cost pulls the rug out from under the feat of potential attackers. It effectively makes decompilation of the code unattractive because it is not worth the effort to recover the source code from the program by decompilation when the original source code can be purchased from the supplier for a fraction of the cost. This idea may reduce the likelihood of decompilation attacks but it defeats the entire purpose of protecting intellectual property against reverse engineering because it involves giving away the intellectual property contained in the source code anyway when it is sold.

Server side execution of code is a very effective method to protect code against decompilation. The application is offered as a remote service where users will connect trough an interface and never gain physical access to the application making reverse engineering of the code very difficult. Unfortunately this is not a solution for systems that have to run in a stand alone environment aboard a naval vessel.

We are interested in technology that can protect code in a way that makes reverse engineering technically so difficult that it becomes impossible or at the very least economically inviable. Some potential techniques that might help in reaching these goals and are discussed in the subsequent sections.

2.3 Bytecode Encryption

The problem of decompilation of Java bytecode is almost as old as the language itself. One technique to protect Java bytecode against reverse engineering attempts is by encrypting the bytecode. The idea of bytecode encryption is to encrypt a classfile and decrypt it just before it gets executed preventing decompilation of the code. The encrypted class-file appears to be protected against disassebling and decompiling attacks but it can be reverse engineered fairly easy indirectly by letting the class loader decrypt and dump the unencrypted class to a stream or file.

The concept is therefore flawed because this type of protection can be easily bypassed by creating such a custom class loader. There is also the problem of key security because the cryptographic key needs to be part of the application.

Simply put, if code is executed in software by a virtual machine interpreter then it is always possible to intercept and decompile the decrypted bytecode [7].

For the JVM this is evident, because it has to adhere to the JVM specification and it has to create new classes according to the Java class-file specification, whereby the Java class-file byte array must contain the unencrypted class definition [12]. Intercepting all calls to the method is all it takes to recover the classes.

Java obfuscation schemes based on bytecode encryption do not work as a protection mechanism because they are easily circumvented with a custom class loader [23]. Given the weak protection and false sense of security this bytecode encryption technology offers it is not considered as a serious protection mechanism.

2.4 Code and Control Flow Obfuscation

As we explained in Chapter 2.2 Java class files can be reconstructed into Java sources that closely resemble the original source with great ease. This is due to the design goals and trade-offs made in the language to achieve compactness, platform independence, network mobility and ease of analysis by bytecode interpeters/JIT.

Obfuscation is the concept of converting a program into an equivalent one that is more difficult to understand and reverse engineer [17].

An obfuscation transformation can be defined as: $P \xrightarrow{\mathcal{T}} P'$ whereby the source program P and the target program P' have the same observable behavior. P' might have side effects that P does not have and they don't have to be equally efficient. In fact most transformations will result in P' being slower or using more memory [7].

Such transformations can be classified and evaluated with respect to their potency (to what degree is a human reader confused), resilience (how well are automated deobfuscation attacks resisted) and cost (how much overhead is added to the program) [7].

A distinction between surface obfuscation (obfuscation of the concrete syntax of a program) and deep obfuscation (obfuscation of the structure of the program e.g., by changing its control flow or data reference behavior) can be made. The latter is considered more difficult to reverse engineer because it requires reasoning about the semantics of a program. The former makes it difficult for humans to understand the source code but does nothing to hide the semantic structure of a program and remains fairly easy to reverse engineer by algorithms used by automatic deobfuscators [26].

Code obfuscation tactics can furthermore be divided into three categories: source code obfuscation trough transformations of source code, Java bytecode obfuscation trough transformations on the bytecode, and binary obfuscation trough binary rewriting [17].

Source code transformations have a few advantages over binary transformations: source code contains more high-level information making it possible to achieve more complex transformations, source code is architecture-independent and obfuscation techniques on source code might blend in better with existing code. There are also some drawbacks such as: transformations can be undone

by the compiler, low-level information is not yet available before compilation and often the transformations are performed per compilation unit as opposed to the entire program.

Binary rewriting has a few advantages over source code rewriting because it allows transformations that require exact addresses and assembly instructions. The obfuscation will also not be undone by a compiler or optimizer because it is the last step in the software development cycle. Disadvantages are the lack of high-level information such as type information. Changed code is under certain circumstances also easier to detect because it can stand out from the surrounding code. The binary rewrites are also architecture-dependent.

Java bytecode transformations are situated between source code and and binary transformations. It is similar to binary obfuscation in the sense that the bytecode contains low-level instructions such as memory addresses for the virtual machine. At the same time it still contains a lot of the source code level information. The presence of source code information can be regarded as a double-edged sword. It makes Java bytecode more susceptible to reverse engineering then binary code from a compiled language, but the additional information could also be used to facilitate obfuscators to produce complex transformations on bytecode that are not possible with binary transformations. Retaining virtually all source code information is necessary because the Java bytecode verifier has to verify the reliability (it checks if the code does not forge pointers, does not violate access restrictions and accesses objects as what they are) of the code. To ensure safe execution of bytecode the following restrictions are imposed that do not apply to binary code: code loaded into the Java interpreter can not modify itself, there can be no operand or stack overflows, types of bytecode instructions should always be verifiable correct and access to object fields have to be legal actions. This limits the degree of obfuscation that can be applied to Java bytecode because the bytecode verifier must still be able to prove the reliability of the transformed code.

Obfuscation techniques applied to Java bytecode can nevertheless prevent certain automatic software analysis tools and decompilers from generating correct source code [19]. Even though the Java bytecode verifier poses restrictions on the bytecode transformations there is enough leeway for obfuscation transformations, because the Java bytecode specification is more expressive than the Java language itself. This makes it possible to produce valid bytecode sequences that have no equivalent source code counterpart.

Code obfuscation is a promising defense technology to secure software in a way that makes the cost of reverse engineering prohibitively high [13].

Code obfuscation has therefore attracted attention as a low cost approach to improve software security by making it difficult for attackers to understand the inner workings of a program. Many obfuscation techniques designed to increase the difficulty of static analysis can however be defeated using combinations of static and dynamic analyses [26].

That obfuscation can not guarantee total protection against reverse engineering has been proven by Barack et al. who informally refer to an obfuscator as a compiler that takes an input program and produces a new program that has the same functionality as the original input yet is unintelligible in some sense. They compare an obfuscated program to a black box and use oracle ac-

cess from computability theory to show that obfuscation is impossible [3]. When their formalizations are even further relaxed to include obfuscaters that are not bound to run in polynomial time and only preserve approximately (agree with high probability on same input but not always) the functionality of the original input program the impossibility still holds. They joke that analyzing programs in rich enough formalisms is hard and sometimes feel as if they have been obfuscated. All programmers know that total unintelligibility is the natural state of computer programs and it is a challenge to prevent a program from deteriorating into that state. Some examples are given to rule out obfuscation for certain applications. Although they prove the standard virtual black box notion of obfuscators as impossible it is also pointed out that it does not mean that there is no method of making programs unintelligble in some meaningful and precise sense.

2.5 Code Virtualization

Code virtualization is a special kind of code obfuscation. Typically for code virtualization is the creation of a custom virtual machine that interprets the custom bytecode generated by the obfuscation process [22]. The original program logic is compiled to the custom bytecode and often integrated within the virtual machine bytecode creating a self contained program that displays similar observable behavior. This makes it difficult to recover the logic of the original program because it is difficult to distinguish between original program instructions and instructions from the virtual machine. Therefore the analysis of the entire bytecode is required. Once the structure and logic of the virtual machine have been recovered the workings of the original program logic must be reconstructed by identifying the instructions hidden inside the virtual machine code. To make things even more challenging some code virtualizers use randomization to make the generated custom bytecodes and virtual machines unique. Randomizing makes pattern matching difficult and can be accomplished by varying the virtual instruction set. This makes reverse engineering harder because a successful attempt does not help to recover another virtualized program generated with different parameters. On top of that also multiple virtual machines can be deployed to make things even more challenging.

Table interpretation is one of the most effective and expensive transformations discussed by Collberg et al. in [7]. This effectively is an early description of code virtualization as it is today. A section of Java bytecode is converted into a different virtual machine code which is executed by a virtual machine interpreter included with the obfuscated application. Several interpreters can be included with an application each accepting a different section of the obfuscated application in a different language. There is however a penalty that slows down the application an order of magnitude for every level of interpretation. The transformation technique should therefore be reserved to the most important code sections only and preferably minimize the total runtime of these sections.

Even the strong protection offered by virtualization-obfuscations is vulnerable to manual attacks. The classical model of an executable protection is a wrapper around an executable. This wrapper is responsible for loading the em-

bedded program that has been obfuscated. Malware is a good example of small code portions that use code virtualization to prevent detection from anti virus scanners. In fact nearly every malware sample nowadays is sheathed in an executable protection which must be removed before static analysis can proceed. Virtualization-obfuscation is considered the toughest of known modern software protections [22].

The task of analyzing malware has been made more difficult due to the increasing use of virtualization-obfuscated malware code. Programs that have been virtualized and obfuscated are difficult to comprehend because they are resistant to static and dynamic analysis techniques. Programs that are protected against manual or automated analysis by virtualization-obfuscation are therefore also difficult to reverse engineer [14]. Static analysis can only analyze the visible bytecode from the interpreter. The custom bytecode of the original program is incomprehensible without reverse engineering the bytecode interpreter and inferring the logic of the byte code program from it first. Traditional static analysis reasons about a program without executing it. Therefore it can only recover the data flow from the encapsulating interpreter.

Such an outside-in approach can not be applied in all cases. There are however also inside-out approaches that aim to identify instructions that affect the observable behavior by complementing existing techniques [8].

An important fact to remember is that code virtualization protected programs do not need to have the property that they are reverted to a fully unprotected state at some point during execution. When code is virtualized it is translated into a different language and executed inside a custom virtual machine which is often obfuscated. The virtual machine has to be reverse engineered before white-box static analysis can be performed. Dynamic debugging is possible but tedious because the traces are dominated by the parsing and dispatching of the virtual machine.

2.6 Evaluating Java Programs

Measuring performance of programs implemented in a managed language such as Java is not a trivial task. There are many factors involved that affect benchmark results, therefore it is important to use an evaluating approach involving statistically rigorous data analysis to achieve more accurate results [10]. Managed runtimes are challenging because there are more factors affecting the overall performance compared to compiled languages. A Java application's performance obviously depends on its input, but on top of that there is the problem of non-deterministic runtime behavior caused by the JVM that could make a program execution differ from run to run. This non-determinism can be caused by the Just-In-Time compiler, a garbage collection that might kick in during a run, heap size that might vary between runs and other optimization tactics applied by the virtual machine. That is why a carefully chosen and well-motivated experimental design in combination with statistically rigorous data analysis is necessary when dealing with non-determinism in managed runtime systems such as Java.

Analyzing software to extract behavioral information from a program is an active field of study. Software can by analyzed by studying models or evaluated

with simulation. Model checking is considered more effective because it checks a property exhaustively in contrast to the tiny subspace tested in an ad hoc manner by simulating [11]. Model checking requires however models that are a good representation of the system under investigation. Dynamic analysis by simulation has however the big advantage that it can be performed with a live running system including the runtime effects such as race conditions caused by threading or distributed components executed on different machines.

Evaluating Java programs protected by code virtualization is slightly more complex. Protected code that has been virtualized can be regarded as a black box hidden inside the program. As already hinted in Chapter 2.5 it would require reverse engineering of the program before white box analysis techniques can be applied and the whole point of code virtualization is to prevent reverse engineering from happening. It would also defeat the entire purpose of evaluating a protected program, because that reverse engineered version would have been reverted to an unprotected state. Dynamic analysis of programs with code virtualization protected sections is however not a trivial task either. Especially when additional control flow obfuscation tactics are applied, such as the introduction of additional branches or adding and execution of non-functional garbage code to conceal the original routines that have been protected. These techniques are deliberately used to mislead and throw-off dynamic analysis techniques and generate misinformation in the traces.

2.7 Threat Levels

There are many scenarios of reverse engineering attacks thinkable varying from negligible threats to highly skilled and motivated adversaries. In order to answer research question 1.5 and 1.6 we introduce the concept of threat levels which allows more nuanced statements.

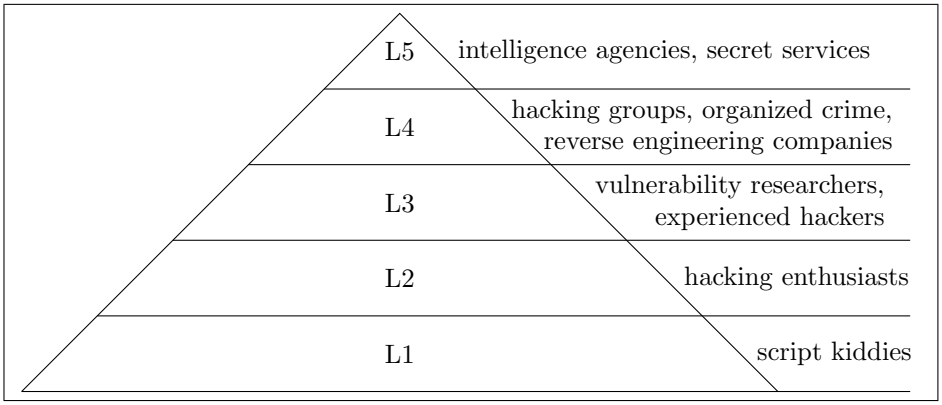

Figure 2.1: Threat Levels Pyramid.

Figure 2.1 illustrates the concept as a pyramid. The height of the pyramid indicates the risk of a successful attack and the width represents the number of adversaries associated within that threat level. The wide base of the pyramid shows that there are many potential attackers on that level but they are not likely to succeed. Every level higher increases the risk on a successful attack,

but the number of attackers with the necessary skills and resources to operate on that level decrease. The top represents the highest risk level with attackers that have the required skills and resources at their disposal to most likely launch a successful attack.

Level 1: The lowest threat level is defined as the category of amateur attackers that pose the smallest risk to successfully reverse engineer an application. The attackers typically have no or low skills and on top of that they lack motivation and resources. Examples are script-kiddies.

Level 2: The medium threat level constitutes the group of individual attackers that have reasonable skills and motivation but lack in the resource department. Many hacking enthusiasts can be assigned to this category.

Level 3: The high threat level consist of groups of highly skilled and motivated attackers with some potent resources. Experienced hackers and vulnerability researchers are good examples that act on this level.

Level 4: The very high threat level with organized groups of highly skilled and motivated attackers that also possess substantial resources. Attackers operating on this level are hacking groups, organized crime organizations, reverse engineering companies.

Level 5: The maximum threat level is represented by highly motivated, disciplined and experienced attackers with the highest skill levels, backed by virtually unlimited resources. Only intelligence agencies and secret services from state actors are capable to operate on this level and they represent the biggest risk because they have access to all the reveres engineering tools, technologies and resources.

2.8 Exploring Solutions for Similar Problems

Theory discussed so far has provided several techniques to protect software. From each of the discussed categories we mention one product and based on empirical evidence relate that to a level of protection in practice. The protection levels are categorized according to the threat level pyramid introduced in Chapter 2.7.

Proguard: This is an open source tool that employs obfuscation to protect Java bytecode. It reduces the program size by removing unnecessary code and enhances performance by optimizing the remaining code [15]. The obfuscation is however limited to weak obfuscation techniques such as identifier renaming and removing of debug information. There is no big cost regarding performance, in fact performance is increased for certain aspects, but the the protection provided is weak and limited to level 1.

Jarcrypt: Also an open source tool, but it utilizes class file encryption to protect the bytecode against decompilation [1]. As already mentioned in Chapter 2.3 this is a weak protection mechanism and easily circumvented. It provides minimal protection limited to level 1 attackers at best.

Wibu: This is a company that offers class encryption technology combined with a hardware component such as a USB dongle [2]. The hardware element offers some additional security improvement over software based solutions. There are however weaknesses in the technology that can be exploited, therefore the degree of protection is limited to level 2.

Although the three products mentioned above and their competing rivals have been around for several years they fail to protect Java bytecode sufficiently against reverse engineering. Until recently there were no capable solutions available on the market that offered any kind of protection against the higher threat levels. Currently there is a promising technology that uses code virtualization as the primary protection technology called Solidshield. This technology has our interest. It is however relatively new and fairly unknown. This thesis investigates therefore this solution and discusses it in Chapter 5. The results from evaluation the protection are presented in 8.

2.9 Recap

The lessons learned from literature so far recapitulated:

- Literature has shown there are several use cases for reverse engineering such as software maintenance, understanding legacy code, generating models, understanding protocols, detecting malicious programs, etc. Reverse engineering is however also applied to uncover intellectual property like algorithms, trade secrets, etc. from software.
- Java is very susceptible to reverse engineering because the class files containing the bytecode retain a lot of high-level information from the original source code.
- Techniques such as obfuscation have been devised against reverse engineering of software. It has however been proven that it is impossible to obfuscate programs in a way that can not be deobfuscated [3][5].
- Code virtualization is an interesting technique to increase the effort required to reverse engineer the code because in order to understand the virtualized bytecode the corresponding virtual machine must also be decompiled and analyzed.
- Given enough time and resources every system could be reverse engineered. Only by significantly increasing the required time and or effort, making it hard to reverse engineer a system, one might make it an uneconomical undertaking enough to greatly reduce the attempts.

- Gathering quantitative measurements are mandatory for our research where we intend to compare different implementations to each other. There are several areas of interest and variables that have to be taken into account.

In Chapter 8 we investigate combining the code obfuscation and code virtualization technologies by benchmarking them.

Chapter 3

Testing and Evaluation Methodology

In order to collect meaningful measurements from Java it is important to use a robust evaluation approach. This is notoriously difficult as we mentioned in Chapter 2.6. Obfuscation and code virtualization protection reduce the available options for analyzing protected programs. Here we motivate our approach to collect and analyze data in a statistically rigorous manner.

The approach is applied to test the sorting program from Chapter 4 and the TACTLESS demo application discussed in Chapter 7. Both programs implement a set of sorting algorithms that are benchmarked to evaluate the Solidshield code virtualization tool discussed in Chapter 5.

3.1 Benchmarking

Java is a managed programming language which means that the inherent non-determinism caused by the JVM has to be taken into account when taking measurements. Factors specific to Java are i.e. the Just In Time Compilation (JIT) compiler that can cause a program to execute differently when executed multiple times and the garbage collection that affects memory allocation. There are also non-Java-specific processes that could affect measurements that are platform- or hardware-related. The former could be operating system dependent while the latter might be caused by generated system interrupts or multi-threading processors, etc.

The non-deterministic behavior can lead to variability in the measurements. To prevent skewed results an approach to address these issues is needed. For the sorting program benchmark we are primarily interested in execution times of the algorithms to relate them to time complexity. For the TACTLESS demo application we included other metrics such as stack size, thread count, etc. and relate this to space complexity. The results of the non-obfuscated sorting algorithms are compared with the obfuscated and virtualized algorithm results. These results are indicative for the suffered performance deterioration.

The following benchmark strategy has been devised to minimize the execution time variability and heap size variability during our experiments with the

sorting algorithms.

We are interested in steady-state performance. This can be achieved by running the programs for several iterations inside the JVM and by timing the execution time we can calculate a mean or median with high confidence. For more reliable results the max and min values have been taken out of the equation for calculating the arithmetic mean. Start-up performance is included for completeness. It is however advisable to ignore the first few iterations when pollution of the measurements with JVM initialization, class loading and JIT compilation related overhead is undesirable.

All sorting algorithms are called from a single benchmark class and executed inside one JVM. Bottlenecks are removed by loading the data sets into memory before the algorithms are called. This ensures that they are not limited by external factors such as disk or network throughput. For every new iteration the garbage collector is called to clean garbage from the heap. Although it is not possible to force garbage collection in the JVM, hinting the garbage collector before executing a new run is still considered beneficial in reducing non-deterministic behavior resulting from garbage collection during a run. The first iteration of every test is discarded. The subsequent thirty iterations are timed and these runtimes are recorded to a file. These results are used to calculate mean, median and the standard deviation. The sorted elements are also stored in a file on disk and the md5hash is compared to a reference file that has been sorted. When the hashes don't match this would indicate wrong sorting results which might imply that the obfuscation or virtualization produced code with behavior that wasn't functionally equivalent to the original unprotected sorting algorithm used as reference.

This is one test procedure. Benchmarking however consists of executing the test procedure several times creating a new JVM for every test procedure execution. Combining the results give an overall result with higher confidence.

The tests are performed on a quad core Intel Xeon E5-1620 v2 at 3.7 GHz with 16GiB RAM running on CentOS 6.7 with a Linux kernel from the 2.6 branch.

The approach applied to the sorting bundles is very similar. An adapted version of the testing class has been converted into a sorting tester bundle. There are however a few details slightly different. Instead of relying on reflection to load the sorting algorithm classes the declarative services model is used to dynamically load and wire a sorting algorithm bundle to the testing bundle. These declarative services add additional non-determinism. There are also more classes and objects loaded into memory during the bundle tests, because the OSGi framework requires the necessary components to run. The statistically rigorous approach is however designed to adjust for these conditions. The repetitive testing in multiple iterations smooths out variations and compensates for deviations caused by external sources, such as loading of classes by OSGi and the component model. When the variation is too big this will become apparent in the calculated standard deviations.

3.2 Metrics

For the sorting algorithms the benchmarking approach determines the amount of time required as a function of the data input size. There are however several aspects that contribute to the overall performance of an application. The CPU load and memory consumption are important factors. Given the nature of the JVM it would be good to have more detailed information regarding the memory usage. Furthermore concurrency is an important aspect that should be considered in an evaluation. Therefore the following metrics have been selected:

1. CPU Usage
2. Memory Usage
    - Heap Memory Usage
    - Non Heap Memory Usage
    - Memory Pool Old Gen
    - Memory Pool Eden Space
    - Memory Pool Survivor Space
    - Memory Pool Meta Space
    - Memory Pool Code Cache
    - Memory Compressed Class Space
3. Threads
    - Number of Active Threads
    - Peak count
    - Number of Daemon Threads
    - Total Number started

These metrics are collected trough a Java Management Extensions (JMX) agent that can connect to the JVM that runs the programs under investigation. The JMX agent can connect to a remote JVM running on a remote system on the network. This is advisable to minimize the system load for the machine executing the test.

Chapter 4

Sorting Algorithms

The experiments carried out in Chapter 8.1 have been performed on a small sorting program. This program has been used for benchmarking experiments. We selected three sorting algorithms that represent the intellectual property that had to be protected against reverse engineering. These sorting algorithms were picked from different complexity classes and have later also been integrated into the TACTLESS demo application discussed in Chapter 7. Sorting algorithms provide an accommodating setting for analysis because they have been documented well in the literature and are therefore predictable. The different asymptotic orders give a good insight in the performance loss inflicted by the code obfuscation and virtualization. The sorting algorithm runtime measurements collected during the benchmarking procedure are used as a reference to compare the performance of versions that have been obfuscated and virtualized by Solidshield.

Besides benchmarking these three algorithms the sorting program also incorporates reflection and threading, two Java language features that are present in TACTICOS. Reflection code to dynamically load and execute tests is used by the sorting program to verify if the virtualization protection does not break the Java reflection mechanism. Threading is tested by benchmarking an adapted version of the sequential quicksort sorting algorithm. This algorithm lends itself better for threading then bubblesort and bucketsort. The benchmark results of the threaded implementation is compared to its sequential counterpart.

The experience gained from these experiments has been used to design the TACTLESS demo application that contains important components from the TACTICOS technology stack. TACTLESS has been designed as a research vehicle to experiment with code virtualization in a more realistic setting that represents the TACTICOS technology stack in a better way. The demo application however also contains the sorting algorithms and some testing code from the sorting program. This allows comparing of the search algorithms results with search algorithm tests run inside the demo application framework to establish the overhead generated by the demo application framework. The TACTLESS demo application design is presented in Chapter 7 and the demo application search results are discussed in Chapter 8.2. This chapter continues by introducing the algorithms and sorting program and concludes with the results gathered from the sorting algorithms.

4.1 Bubble Sort

This sorting algorithm is chosen because of its average and worst case complexity of $O(n^2)$. It is also a simple straightforward algorithm that can be easily analyzed and shown on a single sheet during a presentation. The best case is $O(n)$ and the space complexity is $O(1)$. Maybe not the fastest search algorithm but for the sake of demonstrating performance deterioration expected to occur after obfuscation and virtualization it could be an interesting test. As illustrated in Figure 4.1 the average and worst case complexity grow polynomial. This increase is interesting to measure and compare to obfuscated and virtualized versions of the algorithm. Our bubble sort implementation will be referred to as BubbleSort.

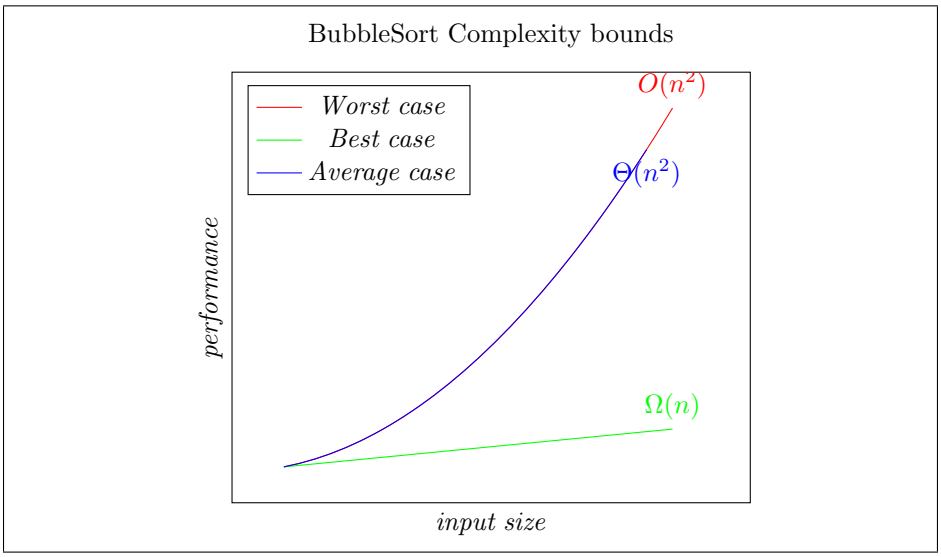

Figure 4.1: Upper and lower bounds of the BubbleSort algorithm.

4.2 Bucket Sort

This algorithm has been chosen because of its complexity classification of $O(n+k)$ best and average case and $O(n^2)$ worst case. The space complexity is $O(n)$. It is a divide and conquer algorithm and the fastest of the three selected algorithms. As shown in Figure 4.2 it operates in linear time and scales well for increasing input sizes. This allows us to see how the Solidshield produced code scales with large data sets. Our bucket sort implementation will be referred to as BucketSort

4.3 Quick Sort

A divide and conquer algorithm. Best and average case are $O(n\,log(n))$ and the worst case is $O(n^2)$. The space complexity is $O(log(n))$. Figure 4.3 illustrates how the time complexity of quicksort is less efficient then bucketsort but on

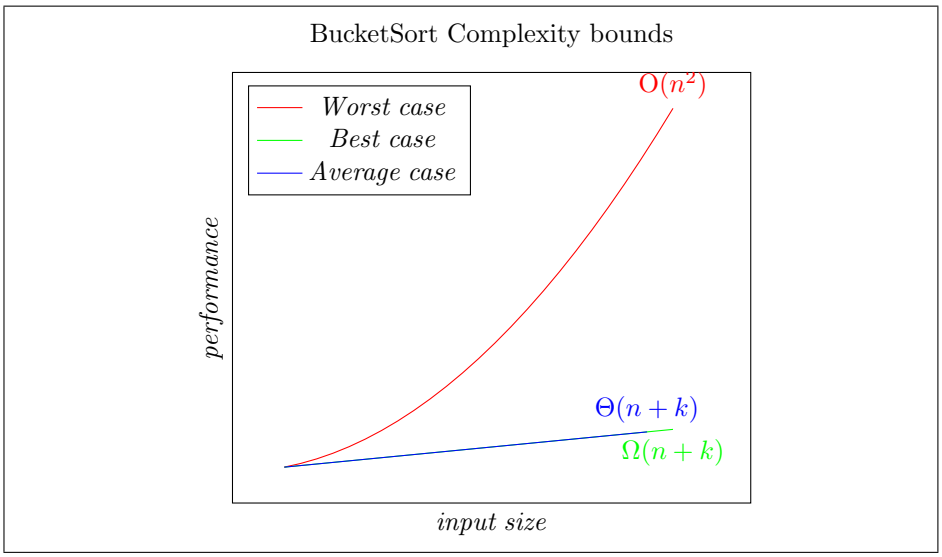

Figure 4.2: Upper and lower bounds of the BucketSort algorithm.

average much better then bubblesort. The algorithm has been implemented as a recursive algorithm as it will be interesting to see how recursion is handled by the obfuscation and virtualization transformations. When they introduce additional overhead such as variables and branches then this could affect the stack severely. Our quick sort implementation will be referred to as QuickSort

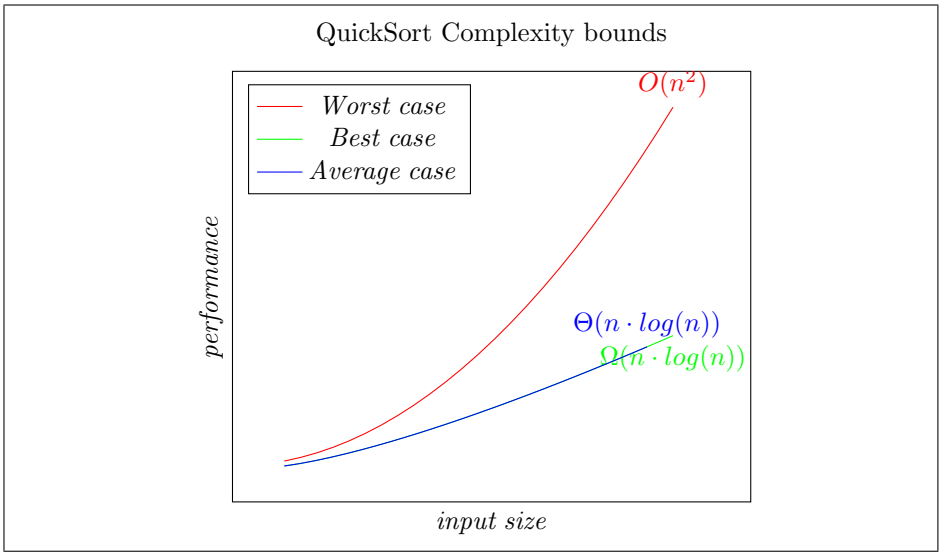

Figure 4.3: Upper and lower bounds of the QuickSort algorithm.

4.4 Sorting Program

For the first phase of our research we have defined a small sorting program to test the performance of the earlier mentioned sorting algorithms when protected by obfuscation or code virtualization. It also served the purpose of testing the compatibility of Java language features and some technologies from components used in the TACTICOS technology stack.

Although there are some benchmark frameworks available for Java we opted to use our own minimalistic test framework, because we want reliable and consistent measurements without the additional overhead introduced by these frameworks. In fact most of these frameworks do not produce accurate results anyway due to some typical Java benchmark pitfalls discussed in Chapter 3.1. There is therefore no need for feature-rich frameworks as they do not produce better results then our testing procedure. They might however influence and pollute the measurements due to the additional required infrastructure to implement them. On top of that it could be an extra source of non-determinism in the sorting program which is something that we would like to avoid.

The runtime measurement should ideally reflect the runtime of the algorithm code and nothing else, but this is notoriously difficult to achieve with Java. We are primarily interested in comparing the sorting algorithm runtimes to each other. Our framework is dedicated to minimize non-determinism and measure elapsed time in a statistically robust way. The measurement resolution is accurate enough to unveil trends we are interested in. Other measurements such as the number of threads, garbage collection, etc. can be extracted from the JVM with JMX. Therefore the choice to collect merely timestamps directly before and after the execution of a sorting algorithm to calculate its runtime seems to be the least intrusive and combined with JMX the most sensible approach. This gives minimal overhead yet reasonably accurate measurements inside the sorting program without unpredictable side-effects and it gives good insight into the operational status of the JVM during the execution.

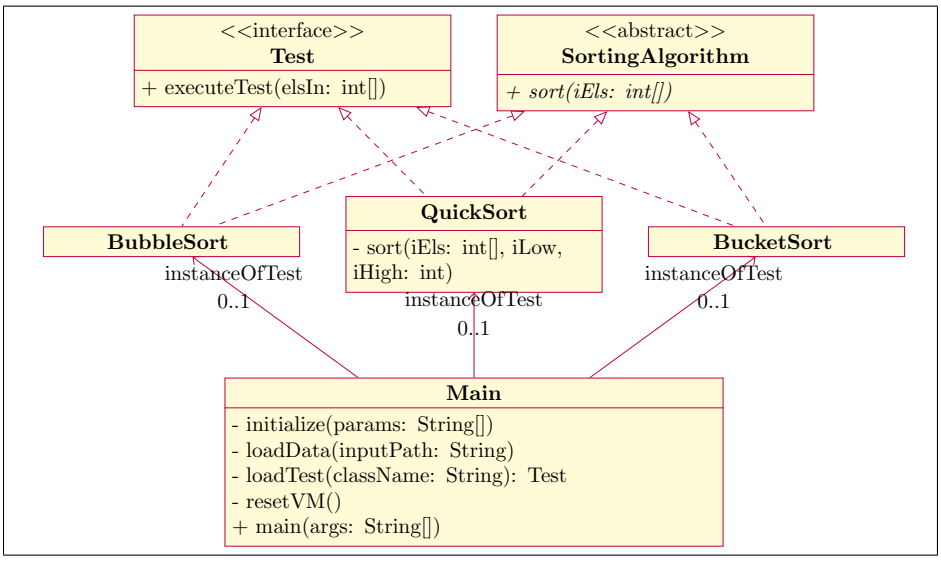

Figure 4.4: UML diagram of sorting program.

The structure of the sorting program is represented in Figure 4.4 as an UML diagram. The main class loads a configuration file based on an Uniform Resource Locator (URL) that is passed as a parameter at start-up. This configuration file contains several parameters such as the locations of the data set files that have to be used, the output path for the files that are written, the kind of tests that have to be executed, options to enable or disable the recording of measurements and to save the sorted results in a file to verify correctness of the output, the amount of test cycles, the number of iterations in one test-cycle, the amount of elements that must be used, etc.

The configuration is loaded and set by the *initialize* method. Once the configuration has finished the program is ready to start the testing procedure as specified in the configuration file. A test is loaded by the *loadTest* method. This method takes a string and uses reflection to return an object of the *Test* type with that name. Then the data set is loaded into memory by the *loadData* method. The data set is then copied and passed to the *Test* object to ensure that the sorting is performed on a new unsorted data set. When the loaded data set is passed by reference the first iteration would sort the data set and all subsequent iterations would run on an already sorted data set. Before performing a test iteration the JVM is hinted by the *resetVM* method to perform a garbage collection twice. This is done to minimize the risk of garbage collection during a test run. Then just before calling the *sort* method of the loaded test a timestamp is taken. Once the sorting algorithm has finished a second timestamp is taken and the runtime is calculated as the difference of these two. If the option has been set the sorted data set is saved to a file for later inspection. The test iteration finishes with removing the sorted data set from memory. Then the next iteration can be performed and these steps are repeated. When all iterations on the data set have been completed the original data set is removed form memory and the next specified data set is loaded to start a new test run. Once all test runs have been completed the Test object is destroyed and a new Test object is loaded and all the above mentioned steps are repeated. This continues until the entire test sequence has finished. In this sorting program the tests performed are sorting algorithms but the program can easily be extended with additional tests such as calculating the Fibonacci numbers for example. When the test implements the Test interface the program can execute it. Then the name of that class could be used in the configuration file when specifying the test sequence and the program would be able to run the test thanks to the introspection feature of the Java language.

Listing 4.1 shows a trimmed version of the original source code of the QuickSort algorithm test class. The code has been kept short but the example should speak for itself. There is an *executeTest(int[] els)* method as required by the implemented Test interface. This calls the *sort(int[] iEls, int low, int high)* method which contains the actual implementation of the QuickSort algorithm. The *exchange(int[] iEls, int i, int j)* is a helper method that swaps two elements.

For the sake of our experiment we pretend that the sorting algorithms in our sorting program are important intellectual property. Unfortunately this intellectual property is easily recovered by reverse engineering the Java bytecode of the sorting program. This bytecode can be decompiled by a Java decompiler and return Java code pretty similar to the original source code.

When we decompile the QuickSort class file containing the Java bytecode of the compiled source code shown in listing 4.1 (i.e. with Fernflower [27] or

Proycon [24]) we can almost completely recover the original source code. The result of such a decompilation attempt is shown in listing 4.2. It is almost identical to the original source code.

The intellectual property or trade secrets in our sorting program are therefore very susceptible to reverse engineering attacks. Additional measures have to be taken in order protect these valuable routines.

```
1  public class QuickSort implements TestInterface {
     private static void sort(int[] iEls, int low, int high) {
3      int i = low, j = high;

5      // Get the pivot element from the middle of the list
       int pivot = iEls[low + (high - low) / 2];
7
       // Split into two lists
9      while (i <= j) {
         while (iEls[i] < pivot) {
11         i++;
         }
13
         while (iEls[j] > pivot) {
15         j--;
         }
17
         if (i <= j) {
19         exchange(iEls, i, j);
           i++;
21         j--;
         }
23     }

25     // Recursion
       if (low < j)
27       sort(iEls, low, j);
       if (i < high)
29       sort(iEls, i, high);
     }
31
     private static void exchange(int[] iEls, int i, int j) {
33     int tmp = iEls[i];
       iEls[i] = iEls[j];
35     iEls[j] = tmp;
     }
37
     public void executeTest(int[] els) {
39     sort(els, 0, els.length-1);
     }
41 }
```

Listing 4.1: QuickSort source code

```
1  public class QuickSort implements TestInterface {
     private static void sort(int[] iEls, int low, int high) {
3      int i = low;
       int j = high;
5      int pivot = iEls[low + (high - low) / 2];

7      while (i <= j) {
         while (iEls[i] < pivot) {
9          ++i;
         }
11       while (iEls[j] > pivot) {
           --j;
13       }
         if (i <= j) {
15         exchange(iEls, i, j);
           ++i;
17         --j;
         }
19     }
       if (low < j) {
21       sort(iEls, low, j);
       }
23     if (i < high) {
         sort(iEls, i, high);
25     }

27   }

29   private static void exchange(int[] iEls, int i, int j) {
       int tmp = iEls[i];
31     iEls[i] = iEls[j];
       iEls[j] = tmp;
33   }

35   public void executeTest(int[] els) {
       sort(els, 0, els.length - 1);
37   }
   }
```

Listing 4.2: Decompiled QuickSort

Chapter 5

Code Virtualization Tool

5.1 Introducing Solidshield

Tages is a provider of technologies for secure packaging of software for distribution. One of their protection tools is called Solidshield. The Solidshield tool can be used to protect important routines in a program. By selecting only the crucial portions a trade-off between security and execution performance can be made. This allows not critical code to run unprotected without the performance penalty inflicted by the protection layers.

There is a new version developed for the defense industry. This tool offers state of the art technology for code virtualization and obfuscation against reverse engineering and strong anti-tampering against malicious code modification.

Due to licensing issues it was not possible to have a copy of Solidshield at our disposal during the research phase. Luckily it was possible to cooperate with a foreign branch of the Thales Group that already had a working Solidshield environment available. This allowed us to perform the tests discussed in this report.

5.2 Dissecting Solidshield

There is not a lot of information available regarding Solidshield. We have analyzed the output Solidshield generated from our sorting program and try to analyze how the tool works by looking at the resulting protected program code. Backed by the information provided and our own findings we discus how it works to the best of our knowledge. The Solidshield protection is based on a fully Dynamic Virtual Machine (DVM). The processing unit of the DVM is a complete virtual processor which is randomly generated each time selected portions of code are protected. The selected code is protected by replacing the original bytecode with a new bytecode representation to be executed on the DVM. This new representation is further protected by obfuscation techniques and can only be run on the DVM it was created for. Tages claims that the substituted code is functionally equivalent to the original code. The performance overhead on

executed protected code is said to be between 10% to 30% depending on the protection options used.

The Solidshield tool can be used to protect programs packaged in a Java JAR file by loading them. The classes inside the JAR become visible and the user can select the methods that need to be protected. The type of protection (encryption or virtualization) can be chosen and in the future the parameters, such as the minimum and maximum instruction variants (the Java bytecode instructions are not translated directly one-on-one to a DVM equivalent but to a set of instructions variants that perform the same task) or the junk-code density to create a haystack around the needle that was chosen to be protected, will become tune-able. The current version for Java is still under development and does not yet support tweaking these parameters but a more mature version for C or C++ binaries has successfully implemented this feature.

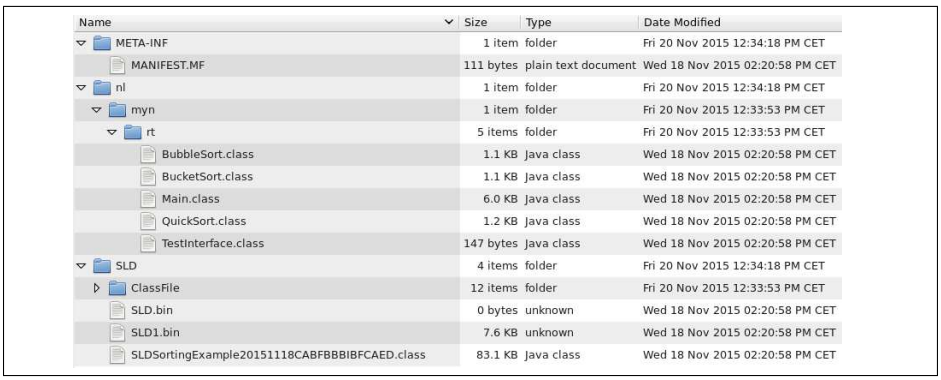

Figure 5.1: Screenshot of a Java JAR archive protected with Solidshield.

The Solidshield protected version of the sorting program shown in Figure 4.4 would result in a JAR file looking like Figure 5.1. Besides the original files in the *nl.myn.rt* package an additional package SLD is added. This SLD package contains two bin files and a class file. The 'SLDSortingExample2015....'.class file is the DVM created by Solidshield. The methods selected for protection are transferred and implemented inside this DVM. The data and DVM instructions are stored in the SLD1.bin file and are loaded upon initialization of the DVM. The subdirectory ClassFile contains the logic and helper classes for the DVM to operate.

When we decompile the class files inside our Solidshield protected JAR file we get the Java code back partially as shown in listing 5.1. That code excerpt shows how Solidshield has made the code less comprehensible.

The *SLD.SLDSortingExa...* import refers to the DVM. Two methods have been protected by Solidshield code virtualization. The *executeTest* method and the *sort* method containing the actual algorithm we wanted to protect. These two methods are now implemented in the DVM. The unprotected *exchange* helper function remains implemented in Java bytecode and is therefore recoverable with decompilation. Due to some level of obfuscation the original variable names have been lost however making it a little bit more difficult to read.

The process of applying the protection can be visualized as Figure 5.2. Ob-

```
import SLD.SLDSortingExample20151118CABFBBBIBFCAED;

public class QuickSort implements TestInterface {
  static SLDSortingExample20151118CABFBBBIBFCAED m_sld = new
    SLDSortingExample20151118CABFBBBIBFCAED();

  public static void sort(int[] var0, int var1, int var2) {
    m_sld.nl_myn_rt_QuickSort_sort(var0, var1, var2);
  }

  public static void exchange(int[] var0, int var1, int var2) {
    int var3 = var0[var1];
    var0[var1] = var0[var2];
    var0[var2] = var3;
  }

  public void executeTest(int[] var1) {
    m_sld.nl_myn_rt_QuickSort_executeTest(var1, this);
  }
}
```

Listing 5.1: Decompiled QuickSort protected by Solidshield

fuscation, encryption and virtualization can be applied to marked methods in a program. These methods are then replaced by proxy methods and the original method body is removed from the class. The virtualized code replacing the original method body is stored inside the data-storage and executed by the DVM both added by the Solidshield tool.

The call transfer taking place in the protected program is schematically illustrated in Figure 5.3. Protected methods are offloaded to the DVM. The original method has been replaced by a proxy method that transfers the call to the DVM. This DVM retrieves the virtualized code and obfuscates it with several obfuscation tactics.

Inside the DVM the virtualized code is i.e. obfuscated trough branching. After loading the virtualized code the main branch is branched into several branches. The branches are randomly selected; each branch contains different code. This process is illustrated in Figure **??**. Besides containing different variations of the virtualized code the branches are also injected with junkcode that is useless but executed with the virtualized code. Flooding the virtualized code with decoy code and randomizing the execution of these branches makes dynamic analysis a very tedious task.

To increase the obfuscation the protected method body does not necessarily correlate to a virtualized method because the DVM can apply code fission and code fusion. This means that the bodies of two or more methods might be fused together in one method as illustrated in Figure **??** and vice versa for code fission where one method body is split over multiple methods inside the DVM.

Besides the before mentioned code obfuscation tactics the DVM can also store the data encrypted inside the data store. After studying disassambled and decompiled versions of our protected programs we found an interesting Exclusive-or (XOR) operation. This XOR was used in the encryption process and we found a pass-phrase required for the encryption and decryption. Encryption on its own is therefore probably not a very effective protection but it

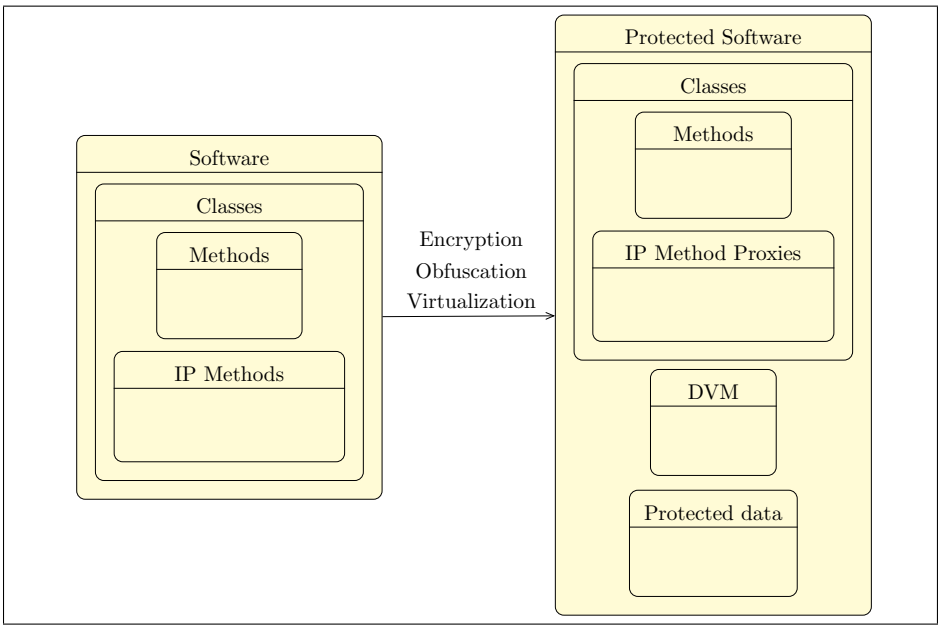

Figure 5.2: Solidshield output.

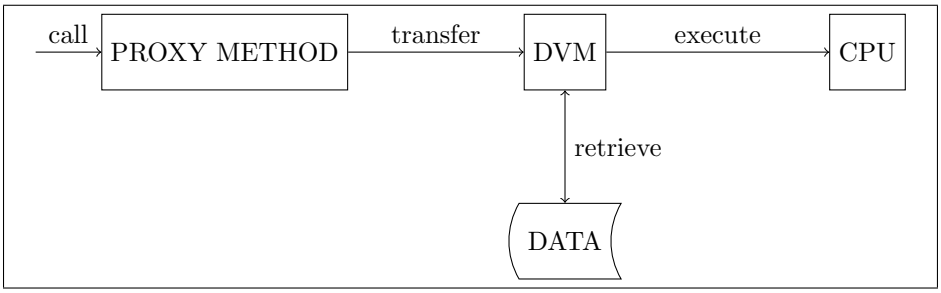

Figure 5.3: Solidshield call transfer.

is a cheap transformation.

It has to be pointed out that decompiling the DVM in a protected program is not straightforward because it contains obfuscated bytecode instructions that can not be expressed in Java source code. Some decompilers can not handle such constructs and fail to decompile the bytecode. There are however a few decompilers that manage to produce an use-able result by leaving these segments in their disassembled state.

To summarize, Solidshield is a tool that employs code virtualization as the primary protection mechanism and on top of that also utilizes some code obfuscation tactics and code encryption. The Solidshield protection approach is not intrusive in the sense that it does not require source code modifications nor do protected applications need a modified runtime environment to execute. Debugging becomes however more difficult on the protected code portions because the original code is lost and the stack traces can therefore not be related to source code.

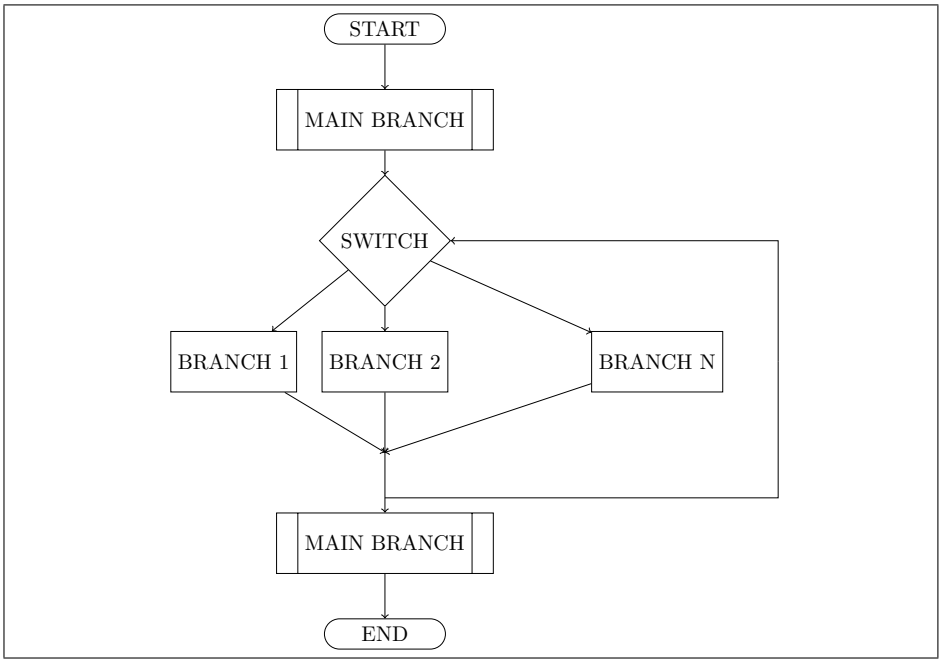

Figure 5.4: Solidshield branching.

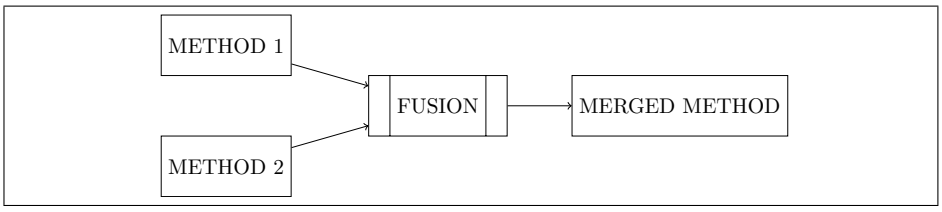

Figure 5.5: Code fusion.

Chapter 6

Technology Stack

The code virtualization tool Solidshield examined in Chapter 5 is tested rigorously with the sorting algorithms presented in Chapter 4. Analyzing the tool is however not enough to determine if it is a suitable candidate for protecting TACTICOS. Applying the code virtualization to TACTICOS to gather performance figures under realistic working conditions is however not possible due to the fact that the Solidshield tool is not internally available. The TACTICOS code base containing intellectual property and trade sectrets is not supposed to leave the Thales premises. Therefore the TACTLESS demo application has been designed to incorporate components from the TACTICOS technology stack. These components are identified and discussed here.

6.1 Java

What is Java?

Java is an object oriented general-purpose programming language designed with the Write Once Run Anywhere (WORA) principle in mind. The cross-platform abilities as proclaimed conform the WORA philosophy allow a Java program to be executed on virtually any platform that has a compatible JVM available without worrying about the underlying computer architecture or operating systems involved. This is achieved by compiling Java source code to the intermediate Java bytecode format that can be interpreted by a JVM. The JVM then translates the Java bytecode to the machine instructions required by the underlying architecture. Java has become a powerful and widely adopted programming language, running on a wide range of hardware platforms ranging from small embedded devices to big super computers and everything in-between.

Where is it used?

The TACTICOS CMS is largely written in Java.

How should it be implemented?

There are some considerations regarding Java that must be taken into account. The current releases of TACTICOS are limited to 32-bit because there are still

legacy 32-bit drivers part of the system. These 32-bit drivers are connected with OpenSplice and limit the communication to 32-bit. Due to the 32-bit limitation there is only a limited amount of memory available for TACTICOS and its technology stack inside the JVM, therefore it is important to know how much memory overhead is generated by Solidshield to ensure that it does not cause 'out of memory' problems.
Although the current releases of TACTICOS still run om Java 6 32-bit this is not a requirement for the demo application. Java 8 and 64-bit would be acceptable, because TACTICOS has to move to 64-bit eventually. The step from 32-bit to 64-bit does not cause many problems. Java 8 is backwards compatible with Java 6 but also introduces some new language features. When a virtualized version of the demo application functions on Java 8 then it will probably also work with the subset available in Java 6. The choice for a newer Java version is therefore justified. Testing and evaluating on a 64-bit JVM also allows bigger data sets to be processed by the sorting algorithm tests and give better insight in the scalability of the virtualization protection. It is however important to know if current TACTICOS versions running on a 32-bit JVM can also be protected without causing unstable and unpredictable behavior due to memory restrictions.

6.2 OSGi

What is OSGi?

Java programs are organized in packages and typically distributed in JAR files. This approach has several drawbacks. The total set of packages constitute the global namespace that has to be linearly searched by the JVM in order to discover and load classes. Java packages offer therefore no real modularity, just merely type checking by the compiler to prevent type conflicts. Bigger and more complex applications also suffer from the additional overhead of a bigger namespace. Another nuisance is the so called JAR-hell, where some part in an application requires a certain JAR but a different part depends on a different JAR. These JARs might be different versions of the same JAR giving problems when their namespace collides or two different JARs that are incompatible because they both require specific versions of a JAR. The problem occurs often in bigger and complex applications because JARs are simply put just a container format without any usable versioning to solve these problems on runtime.
OSGi technology facilitates modularity for Java by introducing the concept of *bundles*. These bundles contain components that typically consist of a tightly coupled collection of classes, JARs and configuration files, packed as normal JAR files but with additional manifest headers. A bundle provides an Application Programming Interface (API) for the services it offers by implementing an interface. The interface is exported while the implementation remains hidden and therefore invisible outside the scope of the bundle. Every bundle is loaded by its own class-loader, localizing the classpath of the bundle and thereby reducing the global classpath size. Bundles can publish themselves as a OSGi service and bind or consume other OSGi services. The dependencies on other bundles are resolved at runtime when the correct versions are wired together. The dynamic component model of the OSGi specification assures interoperabil-

ity between applications and services. Besides modularization, OSGi also offers (remote) lifecycle management for the components including installing, starting, stopping, updating, and uninstalling without requiring a restart of the application.
The OSGi specification has been developed by an open standards organization. This Open Service Gateway initiative has evolved into the OSGi Alliance. The OSGi Alliance is a global consortium consisting of members from very diverse business areas. Initially it started out as a project for service gateways but the focus has broadened to include applications ranging from small embedded devices such as smart phones and other mobile devices, vehicles, consumer electronics, industrial automation, to large-scale distributed systems such as grid computing and enterprise applications. Notable examples include the Eclipse Integrated Development Environment (IDE).

Where is it used?

Complex Java applications such as TACTICOS can be broken down into smaller tightly coupled components with their own dynamic lifecycle management. Besides the obvious modularization advantage brought to Java by OSGi, in terms of reduced complexity by the separation of concerns, the added benefit of simplified deployment and interoperability between components in a standardized environment is also a huge asset. This is demonstrated by the use of Commercial off-the-shelf (COTS) solutions from different vendors that have been integrated in TACTICOS. Logging illustrates this perfectly as an example. There are multiple logging frameworks for Java used in TACTICOS. OSGi allows them to be used interchangeably and seamlessly with other bundles. When a customer has specific requirements regarding the logging or prefers a different framework to be used then the component framework allows this by loading that logging bundle and binding it to the services that require a logging back-end and the dynamic nature of OSGi even allows the logging services to be changed at run-time.

How should it be implemented?

Thales uses the core and compendium specifications of the OSGi 4.3.1 specification. Equinox and Apache Felix are the implementations used for TACTICOS. The compendium adds additional specifications on top of the OSGi core, including the *Declarative Services*. This is a component framework that helps manage the binding of services by declaring dependencies or requirements for bundles in the bundle manifest and component files. The framework handles creation and activation of the components when they are needed by a service that requires them. Declarative services remove the need for adding boilerplate code by moving the OSGi related code from the source code to a higher abstraction level in a declarative style Extensible Markup Language (XML) format. Newer versions of the OSGi specification also support an annotation based declaration that can be used to annotate the source code, but that is not supported in the version used for TACTICOS.
The OSGi bundles each have their own class-loader and the dynamic services rely heavily on dependency injection. Both depend on advanced Java language features such as reflection. It is important to know if the Solidshield technology is compatible with reflection and dependency injection with the multiple class-

loaders used by OSGi bundles.
The dynamic component model which allows dynamic loading, starting and stopping of bundles on runtime should be implemented in the demo application and include Declarative Services. The demo application satisfies this request with a design that utilizes the OSGi principles in a practical way by dividing all selected technology stack components into dedicated bundles. This also allows for the creation of multiple versions of JARs that can be loaded based on configuration files or at runtime from the menu. The ability to load and manage the lifecycle of an unprotected version of a component and several combinations of protected versions of that same component is very useful. This allows for testing permutations of these combinations and debug potential problems without breaking the entire application when there are minor changes in one component that do not affect other components. Especially the fact that this demo application is designed to test and evaluate the Solidshield technology that is not in-house available makes it a very valuable feature. The virtualization has to be applied by an external party that does have a functional Solidshield setup, therefore it is very beneficial when only one bundle hast to be re-protected after a code change instead of re-protecting the entire application. On top of that it is also very practical to have the flexibility to introduce new bundles on a later moment when new functionality is required.

During the development process Equinox can be used, because it integrates perfectly with the Eclipse IDE. In fact the Eclipse runtime is based on Equinox. In theory everything developed in Equinox should be compatible and interchangeable with Apache Felix. This is evaluated by executing the stand alone bundles inside Apache Karaf which uses Apache Felix underneath. Testing and evaluation is performed stand alone inside Karaf, because protected bundles can not run inside the Eclipse runtime.

6.3 OpenSplice

What is OpenSplice DDS?

The Data Distribution System (DDS) for Real-Time Systems is an Object Management Group (OMG) machine-to-machine (m2m) standard that aims to enable scalable, real-time, dependable, high-performance and inter-operable data exchanges between publishers and subscribers via a Data Centric Publish Subscribe (DCPS) architecture. OpenSplice is a realization of this OMG standard and aims to reduce the complexity of real-time distributed systems by providing an infrastructure for building fault-tolerant systems.

OpenSplice DDS has originally been developed as SPLICE-DDS by Thales Naval Netherlands. The first incarnation of SPLICE was designed to be the information backbone in the TACTICOS CMS and as such deployed in over eighteen navies around the world. The second generation COTS evolution of this successful middleware is now available as OpenSplice DDS developed by PrismTech.

Where is it used?

Thales needed a way to exchange real time messages between the distributed information systems in their CMS. These CMSs contain many sub-systems including legacy components. Linking these sub-systems and components together requires therefore technology that supports multiple programming languages and cross platform communication. The requirements for high performance, fault-tolerant, non-volatile information exchange with a small footprint could not be met by existing technology. Thales took therefore the initiative to develop SPLICE-DDS to satisfy these requirements. OpenSplice is adopted for newer systems. Thales developed a back-end library to assure compatibility between the originally SPLICE-DDS based API calls and the new OpenSplice implementation.

How should it be implemented?

OpenSplice is used in the TACTICOS CMS to share data, events and commands between nodes. The nodes can produce information by creating topics and publish samples of these topics. DDS delivers them to subscribers that are interested in that topic. OpenSplice usage in TACTICOS is analogous to a relational database. A topic can be compared to tables in a relational database. Topics can have several types and a number of primary/foreign keys. During the development process the Interface Definition Language (IDL) pre-processor is required to generate topic types and type-specific readers and writers. This IDL pre-processor tool can generate code for C,C++,C# and Java.
The demo application should use several topics types and exchange them between the loosely coupled components. The message size and update rate in TACTICOS typically depend on the source. This might be a sensor with a steady data stream, a receiver tracking plane or ship transponders often have a variable number of tracks depending on the activity in the vicinity or radar that has an update interval of one dish-rotation, etc. The demo application should therefore have configurable settings to vary parameters such as message length, update frequency, number of tracks per update.
The implementation proposal comprises of one OpenSplice bundle that contains the specification, configuration and initialization logic to utilize OpenSplice. Other bundles that require OpenSplice for exchanging topics depend on the OpenSplice reader and writer service provided in that bundle. There is a separate simulator bundle that generates data to be distributed by OpenSplice topics. The simulator bundle mimics several sensors and implements a simplified version of their respective protocols. The sensors communicate via Netty to the simulator bundle and the simulator bundle publishes this via OpenSplice where other bundles such as the Graphical User Interface (GUI) can receive and process them.

6.4 JNA / JNI

What is JNA / JNI?

Java Native Interface (JNI) is a framework that allows a program running in the JVM to call native code and libraries and vice versa. The Java Native Access

(JNA) is a library that provides easy access to native shared libraries. With JNA it is possible to take advantage of native platform features without the overhead of JNI.

Where is it used at Thales?

Both JNA and JNI are used by some of the third party components in the technology stack.

Where should it be implemented?

It is not being implemented yet because it is considered to be very low priority to the project. It might in fact be irrelevant because most of the external libraries are already part of the public domain. If it is used to load drivers or legacy code then it is the question if that could be protected anyway, because that is outside the scope of the virtualization protection applied inside the JVM.

6.5 GStreamer

What is GStreamer?

Gsteamer is an open source multimedia framework written in C that allows a programmer to create components for audio and video playback, recording, streaming, editing, etc.

Where is it used?

The GStreamer multimedia framework is used for processing video streams in TACTICOS such as streaming radar video from the CMS to the GUI.

How should it be implemented?

The framework is written in C and Java code virtualization is therefore not applicable nor is it desirable because it is already part of the public domain. Protecting the video data is also not part of this project. The focus lies on protecting algorithms and evaluating the performance and possible side effects it might have on other processes. It would be enough to have one bundle that streams a video file and display that on a tab in the GUI.

6.6 JOGL

What is JOGL?

The Open Graphics Library (OpenGL) is an API written in C for hardware accelerated rendering of 2D and 3D graphics. Java OpenGL (JOGL) is a wrapper

library around OpenGL that allows OpenGL to be used in Java programs.

Where is it used?

JOGL is used in TACTICOS to take advantage of the hardware accelerated functionality that OpenGL compatible video cards offer e.g. to draw the Tactical Display Area (TDA) and some other components in the GUI.

How should it be implemented?

Being an open source library that is part of the public domain makes protecting its code irrelevant. The main interest here is to validate if JOGL is not affected by side effects in terms of performance overhead or instability created by the code virtualization. This could be achieved by the visualization of a graphical object. Displaying a 3D object in the GUI that transforms (rotates) based on transformations generated from a generator suffices.

Remarks

JOGL uses JNI to access the OpenGL C API.

6.7 SLF4j (PAX logging)

What is SLF4j?

The Simple Logging Facade for Java (SLF4J) is basically a facade pattern that provides decoupling from the Java logging backend. The actual log backend is determined at runtime. It allows several logging backends such as log4j, logback, tinylog, etc.

Where is it used at Thales?

Proper logging is crucial to determine what causes undesired behavior and to solve problems. Especially for military grade systems such as TACTICOS it is important to have accountability. Therefore logging has been integrated almost everywhere inside TACTICOS. SLF4j offers some flexibility by choosing between different logging backends.

How should it be implemented?

There are multiple logging backends used inside TACTICOS. Currently SLF4j version 1.7.5 is used with the PAX backend. The PAX backend uses the Apache log4j logging backend and extends the standard but minimalistic logging interface with additional interfaces. The demo application should contain a logging bundle that provides logging functionality to the application. This bundle should use an OSGi logging service. Then the actual logging service can be

changed and varied without altering the demo application simply by installing a bundle in the OSGi runtime that provides the desired logging service such as PAX.

6.8 Netty

What is Netty?

This is a client-server framework for the development of Java network applications such as protocol servers and clients. It provides asynchronous event-driven network tools to simplify network programming.

Where is it used?

There are many devices such as radars, Automatic Dependent Surveillance - Broadcast (ADS-B), Automatic Identification System (AIS), sensors, etc. that generate data to be processed by TACTICOS. These point to point interface product connections are handled by Netty.

How should it be implemented?

Netty 3.10.4.Final is used by TACTICOS. There is an OSGi bundle available that can provide Netty to other bundles in an OSGi runtime. This should be used by the TACTLESS demo application. Several protocols should be simulated such as ADS-B, AIS and a generic parameterized protocol. A simulator with 5 channels generating a data stream of updates between 300 to 500 tracks/second with an appropriate message size for the respective protocols would be a good representation.

Remarks

There are many systems and protocols that can be simulated with the parameterized protocol without implementing the actual protocol such as e.g. the Nav-radar which uses the National Marine Electronics Association (NMEA) protocol. It generates roughly about 300 tracks per seconds in a Comma Separated Values (CSV) format. The updates are actually occurring every two seconds or so, because the radar rotates and updates on every rotation.

ADS-B and AIS are systems to track respectively flight and shipping movements. Information such as ID, cargo, course/heading, destination, possible danger, etc. are transmitted by a transponder which can be received with a receiver.

6.9 GNU/Linux

What is GNU/Linux?

GNU/Linux is an open source operating system.

Where is it used?

TACTICOS and the JVM require a host environment to operate.

How should it be implemented?

TACTICOS is tested and distributed with hardened kernel versions that have been patched and adapted. It must work with these specialized versions obviously. This is trivial to test.

Chapter 7

TACTLESS Demo Application

The TACTLESS demo application has been developed as a research vehicle that contains components from the technology stack mentioned in Chapter 6, but is itself free from proprietary TACTICOS code. This allows sharing and distributing the TACTLESS code with third parties without risking exposure of intellectual property or trade secrets. TACTLESS components are developed as OSGi bundles and the sorting algorithms from the sorting program discussed in Chapter 4.4 have been adapted and included as sorting bundles. This allows for a performance comparison between OSGi and non-OSGi code virtualization.

7.1 Bundles

The demo application adheres to the OSGi principles. This serves two purposes: implementing and integrating the OSGi requirement identified in the TACTICOS technology stack as well as taking advantage of the increased flexibility offered by OSGi for testing and evaluating the demo application. By choosing this approach we intended to increase the modularity of the demo application to address the problem of being dependent on an external party to apply Solidshield protection to our code. This was necessary because we did not have a Solidshield setup at our disposal during the term of this project. In fact the outcome of our work should lead to a recommendation that will be used to determine whether Thales should consider investing in the Solidshield technology. In the mean time we had to rely on a Thales department in France where they have a Solidshield setup. They would apply the code virtualization to our program bytecode. The development work-flow involved therefore sending our JARs to France which would then be returned protected with Solidshield. This work-flow poses the following problems. Firstly changes or new additions to the program can not be subjected to testing and evaluation directly because they have to be send away to be protected. Sometimes the returned protected version is corrupted and then the process has to be repeated wasting even more precious time. Secondly the time that passes can not be used to test and evaluate other unaffected parts of the program in the meantime because the Solidshield protection is applied on JAR level. This means that a minor change in one class

requires the entire JAR to be re-protected from scratch.
The sorting program as presented in Chapter 4.4 was packaged in one JAR. Figure 7.1 shows a simplified representation of the sorting program. It shows that the classes all belong to the same package. This package is distributed in one JAR. In this situation a minor issue with the Solidshield protection and one of the classes would require adapting the original source code, exporting the entire program to a new JAR and sending it away to be protected. Even when the problem occurs in an unprotected portion of the code this can not be adjusted in a elegant way.

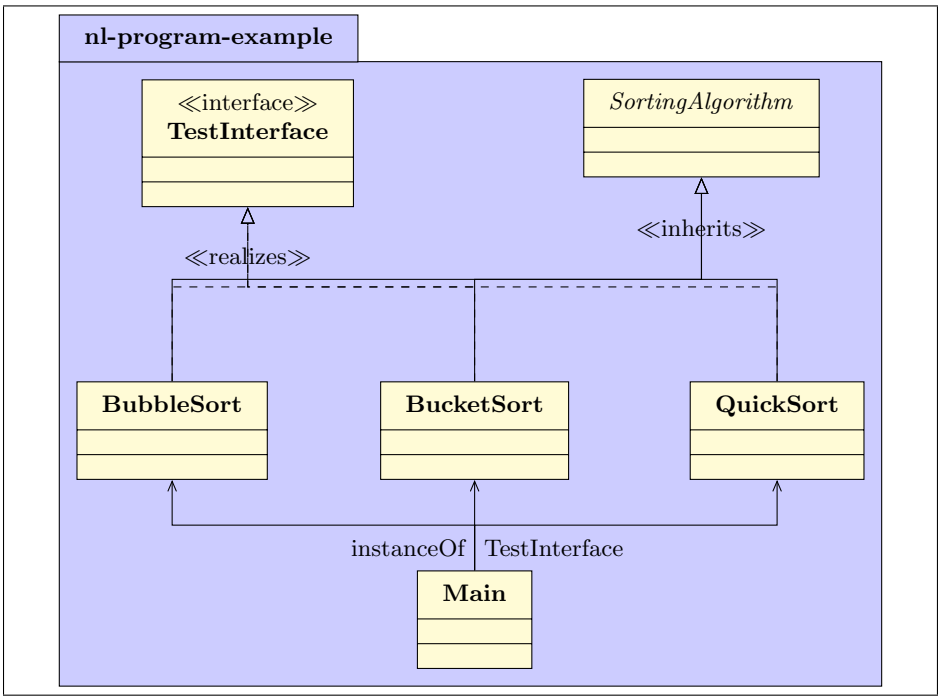

Figure 7.1: UML diagram of sorting program.

The solution to this problem is adopting OSGi bundles to modularize the program. These bundles are basically JAR files with additional meta-information. When the Solidshield protection gives problems in one of these bundles, then only that particular bundle has to be resend to France and in the mean time the testing and evaluating on the remaining bundles can continue. An other advantage is the fact that bundles can have their own lifecycle. This allows bundles to be started and stopped dynamically. We exploit this feature to experiment with different versions and variations of the bundles. In case of a faulty bundle the evaluation does not have to be postponed until a new protected version becomes available because a different version of the bundle can be loaded in its place. This also opens up a wealth of possibilities such as testing different combinations of bundle versions.

7.2 Component

A component is a service provided by a bundle. One bundle can contain multiple components but every component has its own implementation class. The component can provide a service by implementing a public interface and exporting it. Components can also use services provided by other components. Figure 7.2 shows a basic component diagram. The component shown provides a service *A* and requires a service *B*.

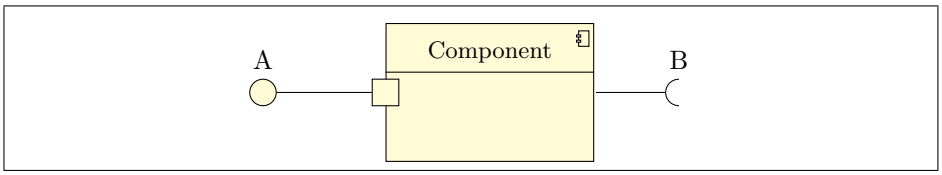

Figure 7.2: UML diagram of sorting program.

Transforming the sorting program into an improved OSGi version to address the problems identified before resulted in a version consisting of eight bundles. As illustrated by Figure 7.3 each sorting algorithm has become a service provided by their respective component. These sorting services are required by the SortingTester component. The component framework binds these together on runtime. Not shown in the simplified component diagram are the *Logging* bundle, the threaded versions of QuickSort and the bundle that actually uses the *ISortingTester* interface.

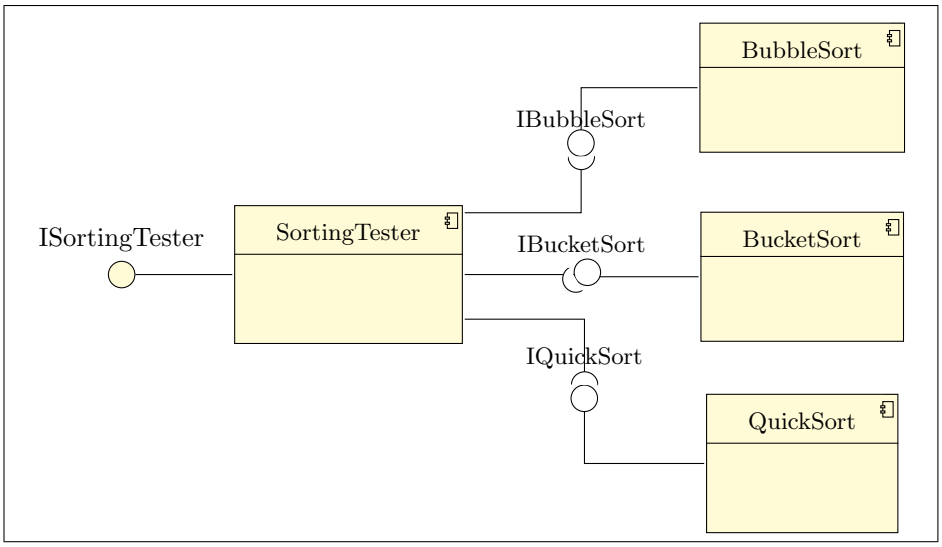

Figure 7.3: Component diagram of sorting program.

Now with this increased modularity a problem occurring i.e. the BubbleSort bundle does not hinder testing with the other sorting algorithm bundles. Testing with the other bundles can therefore continue while we wait on the fixed bundle to be re-protected by Solidshield. Furthermore this also allows us to leave non-critical bundles unprotected. The sorting algorithm bundles contain intellectual property that was selected for protection. The logging bundle and SortingTester

bundle however do not. By leaving these bundles unprotected we can adapt them easily and use that adapted version right away without having to send them away.

7.3 Structure

Figure 7.4 shows a component diagram of the TACTLESS bundles used during the experiments discussed in Chapter 8.1. A larger version of the diagram is included in the appendices as Figure A1.

The TACTLESS bundle requires two services that provide implementations for the INetworkSimulator interface, ISortingTesterInterface. The ISortingInterface is provided by the sorting.test bundle which implements the interface and the testing logic required by our benchmark approach. This sorting.test bundle requires several sorting interfaces that are implemented by the respective sorting algorithm bundles. Every algorithm has its own interface implementation for convenience, because that allows multiple sorting bundles to be active and bound to the sorting.test bundle. The sorting bundles contain only the implementation for the algorithm and have no external dependencies on other services such as logging to reduce non-determinism.

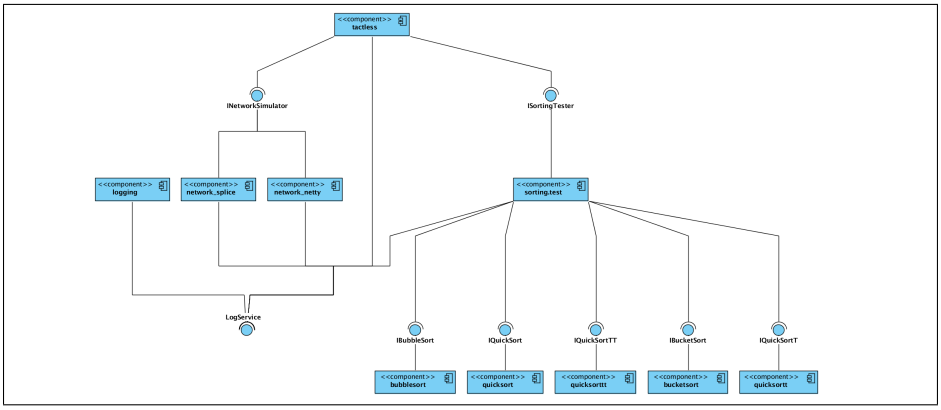

Figure 7.4: Component diagram TACTLESS.

Chapter 8

Results

The results presented here are categorized in two categories. Benchmarking and evaluation of sorting algorithms in a sorting program and evaluation of TACTLESS demo application bundles, including the sorting bundles that contain the sorting algorithms used in the sorting program. These sorting bundles are used to establish baselines and help to determine the influence of an OSGi environment on the code virtualization.

Chapter 8.1 contains the results from the experiments carried out on the sorting algorithm program. The sorting algorithm benchmarks are presented in Chapter 8.1.1 to 8.1.3. Benchmark results of the threaded sorting algorithms are presented in Chapter 8.1.4. Reflection is discussed in 8.1.6.

Chapter 8.2 is dedicated to the evaluation of code virtualization on TACTLESS in an OSGi environment. First Chapter 8.2.1 guides trough the migration from the sorting program towards the TACTLESS demo application running in an OSGi environment. Chapter 8.2.2 purpose is to elucidate the effect of migrating towards an OSGi environment and compares the sorting program results with the sorting bundle results. Finally a TACTLESS example is given with a networking bundle in Chapter 8.2.3.

8.1 Sorting Algorithms

The sorting program introduced in Chapter 4.4 has been benchmarked following the method described in Chapter 3.1. Three versions have been subjected to this benchmark: the original reference version, an encrypted version and the version with virtualized code portions. The encrypted and virtualized versions were created with the Solidshield tool described in Chapter 5. The results of the taken measurements are presented here.

8.1.1 BubbleSort

Table 8.1 shows the measured runtimes of the original bubblesort and the encrypted version of bubblesort. As expected the bubblesort algorithm does not scale very well for increasing data sets. Sorting a data set with 100.000 elements

takes over 12 seconds but sorting a data set of 1.000.000 elements already takes roughly 1252 seconds (over 20 minutes).

	Reference		Encrypted		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$5.83 \cdot 10^{-5}$	$1.1 \cdot 10^{-5}$	$1.34 \cdot 10^{-2}$	$9{\cdot}10^{-3}$	229.9
$1 \cdot 10^2$	$3.72 \cdot 10^{-4}$	$1.6 \cdot 10^{-3}$	$1.99 \cdot 10^{-4}$	$7.8 \cdot 10^{-5}$	$5.4 \cdot 10^{-1}$
$1 \cdot 10^3$	$9.64 \cdot 10^{-4}$	$8.9 \cdot 10^{-6}$	$1.13 \cdot 10^{-3}$	$1.9 \cdot 10^{-4}$	1.2
$1 \cdot 10^4$	0.11	$7{\cdot}10^{-5}$	0.12	$1.6 \cdot 10^{-5}$	1.1
$1 \cdot 10^5$	12.67	$2.1 \cdot 10^{-3}$	12.85	$2.3 \cdot 10^{-4}$	1
$1 \cdot 10^6$	1,252.2	$4.1 \cdot 10^{-1}$	1,267.3	$3.5 \cdot 10^{-4}$	1

Table 8.1: Encrypted BubbleSort measurements.

When we compare the runtimes of the two versions they appear to perform very similar. The encrypted version does not seem to suffer big performance penalties for the added encryption. For the small data sets of 10 elements the performance factor stands out, but this is due to an anomaly encountered in iteration 8 as shown in the scatter plot of E25. This skews the calculated average and therefore the calculated factor for that particular test-set is not representative of the real performance loss. After correcting for the anomaly we calculated a more realistic value of 13 for the performance factor. It appears therefore that the overhead of the encryption is very minimal and only noticeable in the measurements with the small data sets where the runtimes are very short and the decryption-related increase in processing-time takes up a relatively larger portion of the total runtime.

	Reference		Virtualized		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$5.83 \cdot 10^{-5}$	$1.1 \cdot 10^{-5}$	$2.09 \cdot 10^{-4}$	$1.1 \cdot 10^{-4}$	3.6
$1 \cdot 10^2$	$3.72 \cdot 10^{-4}$	$1.6 \cdot 10^{-3}$	$7.25 \cdot 10^{-3}$	$2.4 \cdot 10^{-3}$	19.5
$1 \cdot 10^3$	$9.64 \cdot 10^{-4}$	$8.9 \cdot 10^{-6}$	0.71	$6{\cdot}10^{-4}$	732.6
$1 \cdot 10^4$	0.11	$7{\cdot}10^{-5}$	71.1	$7.8 \cdot 10^{-2}$	640.5
$1 \cdot 10^5$	12.67	$2.1 \cdot 10^{-3}$	NaN	NaN	NaN
$1 \cdot 10^6$	1,252.2	$4.1 \cdot 10^{-1}$	NaN	NaN	NaN

Table 8.2: Virtualized BubbleSort measurements.

Looking at the runtimes of the virtualized version of bubblesort in Table 8.2 indicates that the performance hits are much higher. On a data set with 1.000 elements the reference sorting program took only 0.1 seconds to complete the sorting of the elements while the virtualized version needed 71 seconds to complete the same task. This is an increase factor of 640! Due to the huge performance hits we have not included measurements on bigger data sets because that would require too much time to complete. This test on small numbers does show however that a small processing task can quickly escalate when it is virtualized.

8.1.2 BucketSort

Bucketsort is the fastest algorithm of the selected sorting algorithms. Looking at Table 8.3 we see that the standard deviation for test on 10 elements is too high. The spread is however not an anomaly like we encountered during the bubblesort test. The small data sets should probably not be considered for testing at all because the measurements on the scale of these short runtimes are simply not reliable enough. Looking at the bigger data sets it looks like the encrypted bucketsort performs equal to the original reference program. Only for the smaller data sets the bucketsort algorithm suffers a minor performance penalty.

	Reference		Encrypted		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$6.9 \cdot 10^{-5}$	$1.8 \cdot 10^{-5}$	$2.4 \cdot 10^{-3}$	$1{\cdot}10^{-2}$	34.9
$1 \cdot 10^2$	$7.25 \cdot 10^{-5}$	$1.5 \cdot 10^{-5}$	$9.86 \cdot 10^{-5}$	$2.9 \cdot 10^{-5}$	1.4
$1 \cdot 10^3$	$7.2 \cdot 10^{-5}$	$1.5 \cdot 10^{-5}$	$1.39 \cdot 10^{-4}$	$3.4 \cdot 10^{-5}$	1.9
$1 \cdot 10^4$	$1.45 \cdot 10^{-4}$	$1.6 \cdot 10^{-5}$	$1.64 \cdot 10^{-4}$	$2{\cdot}10^{-5}$	1.1
$1 \cdot 10^5$	$1{\cdot}10^{-3}$	$3.8 \cdot 10^{-5}$	$1.02 \cdot 10^{-3}$	$6.5 \cdot 10^{-5}$	1
$1 \cdot 10^6$	$2.97 \cdot 10^{-3}$	$1.4 \cdot 10^{-5}$	$2.96 \cdot 10^{-3}$	$3.4 \cdot 10^{-5}$	$1{\cdot}10^{0}$
$1 \cdot 10^7$	0.16	$1.3 \cdot 10^{-4}$	0.16	$4.6 \cdot 10^{-4}$	$9.9 \cdot 10^{-1}$
$1 \cdot 10^8$	1.99	$2.1 \cdot 10^{-3}$	1.98	$2.6 \cdot 10^{-3}$	$9.9 \cdot 10^{-1}$
$1 \cdot 10^9$	15.79	$9.7 \cdot 10^{-2}$	16.46	5.8	1

Table 8.3: Encrypted BucketSort measurements.

Virtualization is a different story as can be seen in Table 8.4 where the performance penalty appears to stabilize around a factor hundred. There is a very noticeable anomaly for the test-run on the data set with 1.000.000 elements. The factor is almost five times worse than the performance loss for bucketsort on 10.000, 100.000, 10.000.000, 100.000.000 and 1.000.000.000 elements. At first we suspected a background process or JVM event to be responsible for these odd measurements. Rerunning the test with bucketsort several times on different machines resulted however in the same result. As shown in C4 the spike for the 1.000.000 elements input data set clearly stands out. This seems odd because the performance factor appears to be consistent for the smaller and bigger data sets. We ruled out external factors caused by the machine the test ran on and judging by the standard deviation there are no abnormalities in individual iterations skewing the mean. Taking a closer look on the measurements by interpolating the measurements it appears that the virtualized measurements increase linearly. When the input data set is increased by a factor 10 then the runtimes also increase roughly with a factor 10. This is also true for the reference measurements except for the measurements taken with the 1.000.000 element data set. The reference version of the sorting program sorts that data set much more efficiently than the other data sets while the virtualized version performs as one would expect. The virtualized version does not perform much worse on the 1.000.000 elements, it is just an odd optimization applied by the

JVM for the unprotected sorting program under those specific circumstances. It effectively means that there is not a peak in the required sorting time of the virtualized sorting program but a dip in the sorting time of the reference program. This results automatically in a deviation of the calculated performance factor because that relates the runtime of the virtualized version to the runtime of the reference version.

	Reference		Virtualized		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$6.9 \cdot 10^{-5}$	$1.8 \cdot 10^{-5}$	$1.9 \cdot 10^{-4}$	$1{\cdot}10^{-4}$	2.8
$1 \cdot 10^2$	$7.25 \cdot 10^{-5}$	$1.5 \cdot 10^{-5}$	$4.37 \cdot 10^{-4}$	$6.2 \cdot 10^{-4}$	6
$1 \cdot 10^3$	$7.2 \cdot 10^{-5}$	$1.5 \cdot 10^{-5}$	$2.13 \cdot 10^{-3}$	$7.9 \cdot 10^{-4}$	29.5
$1 \cdot 10^4$	$1.45 \cdot 10^{-4}$	$1.6 \cdot 10^{-5}$	$1.73 \cdot 10^{-2}$	$2.7 \cdot 10^{-5}$	119.3
$1 \cdot 10^5$	$1{\cdot}10^{-3}$	$3.8 \cdot 10^{-5}$	0.17	$6.2 \cdot 10^{-4}$	172.1
$1 \cdot 10^6$	$2.97 \cdot 10^{-3}$	$1.4 \cdot 10^{-5}$	1.72	$5{\cdot}10^{-4}$	578.9
$1 \cdot 10^7$	0.16	$1.3 \cdot 10^{-4}$	17.94	$6.3 \cdot 10^{-3}$	110.9
$1 \cdot 10^8$	1.99	$2.1 \cdot 10^{-3}$	180.29	$2.1 \cdot 10^{-1}$	90.4
$1 \cdot 10^9$	15.79	$9.7 \cdot 10^{-2}$	1,857.53	$6.6 \cdot 10^{-1}$	117.7

Table 8.4: Virtualized BucketSort measurements.

8.1.3 QuickSort

	Reference		Encrypted		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$7.74 \cdot 10^{-5}$	$4.2 \cdot 10^{-5}$	$2.51 \cdot 10^{-3}$	$9{\cdot}10^{-3}$	32.4
$1 \cdot 10^2$	$7.69 \cdot 10^{-5}$	$2.1 \cdot 10^{-5}$	$1.64 \cdot 10^{-4}$	$7.8 \cdot 10^{-5}$	2.1
$1 \cdot 10^3$	$2.35 \cdot 10^{-4}$	$5.8 \cdot 10^{-4}$	$2.75 \cdot 10^{-4}$	$1.9 \cdot 10^{-4}$	1.2
$1 \cdot 10^4$	$8.31 \cdot 10^{-4}$	$3.5 \cdot 10^{-5}$	$1.39 \cdot 10^{-3}$	$1.6 \cdot 10^{-5}$	1.7
$1 \cdot 10^5$	$8.88 \cdot 10^{-3}$	$3{\cdot}10^{-5}$	$1.51 \cdot 10^{-2}$	$2.3 \cdot 10^{-4}$	1.7
$1 \cdot 10^6$	$7.51 \cdot 10^{-2}$	$7.5 \cdot 10^{-5}$	0.12	$3.5 \cdot 10^{-4}$	1.7
$1 \cdot 10^7$	1.14	$1.3 \cdot 10^{-2}$	1.71	$4.8 \cdot 10^{-4}$	1.5
$1 \cdot 10^8$	12.98	$2.4 \cdot 10^{-3}$	18.75	$2{\cdot}10^{-2}$	1.4
$1 \cdot 10^9$	131.22	$1.1 \cdot 10^{-1}$	187.52	10.4	1.4

Table 8.5: Encrypted QuickSort measurements.

Looking at the quicksort algorithm is making things even more interesting. The algorithm has a performance somewhere between bubblesort and bucketsort. It is slower then bucketsort but still fast enough to perform all the tests on the data sets to allow us to compare them to each other. Table 8.6 clearly shows that the virtualized version is about a factor hundred slower then the reference version. The performance factor anomaly for the 1.000.000 element data set is also notable in the measurements presented in Figure C6 but this is also caused by the fact that the unprotected version happens to sort the same data set more efficiently compared to the other data sets. The effect is not as strong as with

	Reference		Virtualized		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$7.74 \cdot 10^{-5}$	$4.2 \cdot 10^{-5}$	$5.48 \cdot 10^{-4}$	$1.3 \cdot 10^{-4}$	7.1
$1 \cdot 10^2$	$7.69 \cdot 10^{-5}$	$2.1 \cdot 10^{-5}$	$9.19 \cdot 10^{-4}$	$6.6 \cdot 10^{-4}$	11.9
$1 \cdot 10^3$	$2.35 \cdot 10^{-4}$	$5.8 \cdot 10^{-4}$	$1.04 \cdot 10^{-2}$	$2.2 \cdot 10^{-2}$	44.1
$1 \cdot 10^4$	$8.31 \cdot 10^{-4}$	$3.5 \cdot 10^{-5}$	$7.39 \cdot 10^{-2}$	$5.2 \cdot 10^{-5}$	88.9
$1 \cdot 10^5$	$8.88 \cdot 10^{-3}$	$3 \cdot 10^{-5}$	0.9	$3.4 \cdot 10^{-4}$	101.6
$1 \cdot 10^6$	$7.51 \cdot 10^{-2}$	$7.5 \cdot 10^{-5}$	9.71	$4.9 \cdot 10^{-3}$	129.4
$1 \cdot 10^7$	1.14	$1.3 \cdot 10^{-2}$	117.39	$5.7 \cdot 10^{-2}$	103
$1 \cdot 10^8$	12.98	$2.4 \cdot 10^{-3}$	1,323.69	$3.4 \cdot 10^{-4}$	102
$1 \cdot 10^9$	131.22	$1.1 \cdot 10^{-1}$	14,246.17	1.5	108.6

Table 8.6: Virtualized QuickSort measurements.

the bucketsort measurements but with the longer runtimes of the less efficient quicksort there is also an anomaly visible for the 1.000.000 data set compared to the otherwise consistent results, just like we saw in the bucketsort results. There appears to be something happening during the sorting that affects the execution time of the 1.000.000 data set much stronger then the other data sets.

The runtimes for the encrypted version produce results similar to the bubblesort and bucketsort measurements. Table 8.5 shows that the performance factor is between one and two. The runtime for the data set with 10 elements is too short for accurate measurements.

8.1.4 Threading

Support for virtualized threading code was one big question mark before we started testing with Solidshield. We adapted the quicksort algorithm and re-implemented it to include threading. Two different implementations have been developed to verify if threaded code could be virtualized and subsequently to determine how this would affect the performance. The initial QuickSortT version gave problems at first when virtualized with Solidshield. The original unprotected version worked like intended, but the protected virtualized version behaved unexpected. During testing we encountered several critical problems such as random lock ups of our sorting program, quitting the program due to an out-of-bounds-exception and wrong sorting results for the bigger data sets beyond a random point. Analyzing and debugging indicated that the problem came from the Solidshield DVM. We discovered the source of the 'array-out-of-bounds' error and shared our findings with the Solidshield development team. They quickly acknowledged the problem and promised a fix within a few days. Roughly a week later we could continue testing. In the mean time we added QuickSortTT, a different and less aggressive threaded implementation of our quicksort test. The primary difference is that QuickSortTT uses explicit synchronization and it utilizes a thread-pool that restricts the amount of threads to the number of available processors on the system the test is executed on, whereas QuickSortT relies on Java Futures and can create more threads then there are processors available.

The newest release of Solidshield included the fix that solved the threading bugs and allowed us to perform our benchmarks with the threaded QuickSortT and QuickSortTT algorithms. Table 8.7 shows the results for the QuickSortT algorithm. When we compare the unprotected reference QuickSortT measurements to the unprotected and non-threaded reference QuickSort measurements from Table 8.6 it becomes clear that threading does improve the performance of the algorithm. Only the small input sets up to a hundred elements perform less and that is because the overhead of dividing the workload over the available cores outweighs the benefit that can be realized on these small input sets.

	Reference		Virtualized		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$1.45 \cdot 10^{-4}$	$3.2 \cdot 10^{-5}$	$8.28 \cdot 10^{-4}$	$7.3 \cdot 10^{-5}$	5.7
$1 \cdot 10^2$	$2.12 \cdot 10^{-4}$	$2.8 \cdot 10^{-5}$	$4.15 \cdot 10^{-3}$	$1.3 \cdot 10^{-4}$	19.6
$1 \cdot 10^3$	$1.68 \cdot 10^{-4}$	$1.5 \cdot 10^{-5}$	$3.68 \cdot 10^{-2}$	$1.3 \cdot 10^{-3}$	219.4
$1 \cdot 10^4$	$3.6 \cdot 10^{-4}$	$2 \cdot 10^{-5}$	0.39	$1.3 \cdot 10^{-2}$	1,091.5
$1 \cdot 10^5$	$2.81 \cdot 10^{-3}$	$1.8 \cdot 10^{-4}$	5.48	$9.3 \cdot 10^{-2}$	1,947.4
$1 \cdot 10^6$	$3.59 \cdot 10^{-2}$	$2.3 \cdot 10^{-5}$	144.87	1.8	4,037.8
$1 \cdot 10^7$	0.27	$1 \cdot 10^{-2}$	576.3	7.8	2,102.2
$1 \cdot 10^8$	3.06	$9 \cdot 10^{-2}$	NaN	NaN	NaN

Table 8.7: Virtualized QuickSortT measurements.

The performance of the virtualized QuickSortT version is however heavily affected resulting in a huge performance factor. For the larger data sets the performance appears to stabilize around a factor of 2.000. The peak for processing the 1.000.000 elements input set, noticed earlier with the other virtualized algorithm measurements, is also clearly present here, but this is caused again by the fact that the unprotected version handles that data set much more efficiently than the other data sets. This huge performance deterioration for the QuickSortT threaded implementation makes it perform worse then the virtualized non-threaded QuickSort for all data sets. The 10.000.000 input set took the virtualized QuickSort 117,39 seconds to sort, but the virtualized QuickSortT needed 576,3 seconds to complete. That is almost five times worse.

	Reference		Virtualized		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$3.33 \cdot 10^{-4}$	$1 \cdot 10^{-4}$	$2.85 \cdot 10^{-3}$	$1.1 \cdot 10^{-3}$	8.6
$1 \cdot 10^2$	$2.74 \cdot 10^{-4}$	$5.1 \cdot 10^{-5}$	$7.84 \cdot 10^{-3}$	$1.2 \cdot 10^{-3}$	28.6
$1 \cdot 10^3$	$2.71 \cdot 10^{-4}$	$5.5 \cdot 10^{-5}$	$1.49 \cdot 10^{-2}$	$1.5 \cdot 10^{-3}$	55.1
$1 \cdot 10^4$	$4.62 \cdot 10^{-4}$	$3.7 \cdot 10^{-5}$	$6.33 \cdot 10^{-2}$	$2.1 \cdot 10^{-3}$	137.1
$1 \cdot 10^5$	$3.13 \cdot 10^{-3}$	$2.9 \cdot 10^{-4}$	0.55	$2.4 \cdot 10^{-2}$	175.9
$1 \cdot 10^6$	$2.8 \cdot 10^{-2}$	$1.3 \cdot 10^{-3}$	7.14	$1.5 \cdot 10^{-1}$	255.4
$1 \cdot 10^7$	0.22	$1 \cdot 10^{-2}$	53.68	1.3	238.9
$1 \cdot 10^8$	2.25	$8 \cdot 10^{-2}$	512.51	7.7	228

Table 8.8: Virtualized QuickSortTT measurements.

Table 8.8 shows that the measurements collected on the unprotected ref-

erence version of QuickSortTT are similar to the QuickSortT reference measurements. QuickSortT performs better on the smaller data sets up to 10.000 elements while QuickSortTT does better on data sets from 100.000 elements and larger. The virtualized version of QuickSortTT however performs much better compared to the QuickSortT version. The performance factor increases with increasing data set sizes and peeks at 255 for the 1.000.000 input set, but the performance factor appears to stabilize around 230 for the bigger data sets. Compared to the non-threaded QuickSort the performance factor increases more, this is however relative to their respective reference measurements. In absolute terms the virtualized QuickSortTT algorithm outperforms its non-threaded QuickSort counterpart for the data input sets we used from 1.000 elements and larger. The reference version of QuickSort took 1,14 seconds to sort the 10.000.000 element data set and the virtualized version completed the sorting in 117,39 seconds which is a factor 103 worse. The QuickSortTT algorithm sorted the same data set in 0,22 seconds and the virtualized version finished the sorting task after running for 53,68 seconds. That is a performance factor of 239. This is a significant performance drop but in absolute terms the virtualized threaded QuickSortTT is still two times faster than the non-threaded QuickSort.

Experimenting with the QuickSortT and QuickSortTT algorithm has shown that threading was not properly implemented by Solidshield. The newest version addressed the issues we encountered and appears to be robust. Based on the measurements it appears however that the implementation of threading does make a difference. The non-virtualized version of QuickSortT was faster then the non-virtualized QuickSortTT for data sets up to 10.000 elements but the virtualized version of QuickSortT was much more affected then the virtualized QuickSortTT implementation. This is an important observation that could imply that developers have to be aware that some coding practices are more efficiently processed after virtualization then others. At this moment it is unknown what causes these remarkable differences. We assumed it might be caused by the more aggressive threading strategy. This could generates overhead and performance penalties for every thread without clear performance benefits if the number of threads surpass the amount of available CPUs on the test system. We falsified that assumption by capping the active thread count to the available number of logical CPU cores on the machine. The problem remained and we needed to extend our test therefore by start gathering additional information from the JVM with JMX, such as thread count and statistics. This gave a better insight on how both threading models are handled inside the JVM. In Chapter 8.2 these metrics are discussed and used to take an other look on the threading issue in an attempt to explain the odd behavior.

8.1.5 Data Set Randomness

The strange peak or valley observed with the 1.000.000 element data set during the sorting algorithm runtime tests were unexpected. Additional tests on ordered data sets have been performed to rule out a problem with the randomness of the randomized input sets. Depending on the algorithm the time complexity might be affected by an unfavorable ordered data set. Although it was unlikely that an unfavorably ordered 1.000.000 data set was the cause for the behavior it had to be investigated nevertheless for completeness. We prove this empir-

ically by applying the sorting algorithms to a randomized, ascending ordered and descending ordered input set.

Table 8.9 shows the results of BucketSort applied to a randomized data set and an ascending ordered data set. The calculated factor indicates that the performance is very similar. This was to be expected because the best and average case both operate in $O(n + k)$ time complexity. The order of elements does not really affect the algorithm execution. Only when the buckets are chosen poorly the algorithm might move to the worst case time complexity but in our implementation that situation does not occur. Table 8.10 shows therefore as expected similar results with the decending order data set.

elements	*random* (*s*)	*ascending* (*s*)	*factor*
$1 \cdot 10^5$	0.89	0.98	1.1
$1 \cdot 10^6$	8.05	7.88	$9.8 \cdot 10^{-1}$
$1 \cdot 10^7$	81.6	78.82	$9.7 \cdot 10^{-1}$
$1 \cdot 10^8$	823.46	788.41	$9.6 \cdot 10^{-1}$

Table 8.9: Protected BucketSort on ascending order data set.

elements	*random* (*s*)	*descending* (*s*)	*factor*
$1 \cdot 10^5$	0.89	0.8	$9.1 \cdot 10^{-1}$
$1 \cdot 10^6$	8.05	8.02	$1 \cdot 10^0$
$1 \cdot 10^7$	81.6	80.3	$9.8 \cdot 10^{-1}$
$1 \cdot 10^8$	823.46	802.98	$9.8 \cdot 10^{-1}$

Table 8.10: Protected BucketSort on descending order data set.

The QuickSort algorithm with its $O(n\ log(n))$ best and average case time complexity is a different story because the choice for an pivot element can impact the runtime significantly. Table 8.11 shows the difference between the sorting runtimes on a randomized set and an ascending ordered set. It appears that an ascending ordered data set is processed roughly a factor 0.6 faster then the randomized input set. This makes sense because when the elements are already ordered the algorithm only needs to compare the elements without utilizing the swap routine.

elements	*random* (*s*)	*ascending* (*s*)	*factor*
$1 \cdot 10^5$	3.37	2	$5.9 \cdot 10^{-1}$
$1 \cdot 10^6$	36.54	21	$5.7 \cdot 10^{-1}$
$1 \cdot 10^7$	424.46	244.56	$5.8 \cdot 10^{-1}$
$1 \cdot 10^8$	4,721.8	2,817.88	$6 \cdot 10^{-1}$

Table 8.11: Protected QuickSort on ascending order data set.

Table 8.12 shows similar results for QuickSort on the descending ordered

input set compared to the randomized set. The key observation to be made here is that the 1.000.000 element factor does not deviate from the other input factors. If that particular data set was not properly randomized then the factor would have to show this but it does not. The peak or dip can therefore not by caused by a poorly randomized data set.

elements	*random* (*s*)	*descending* (*s*)	*factor*
$1 \cdot 10^5$	3.37	2.07	$6.1 \cdot 10^{-1}$
$1 \cdot 10^6$	36.54	22.02	$6{\cdot}10^{-1}$
$1 \cdot 10^7$	424.46	255.37	$6{\cdot}10^{-1}$
$1 \cdot 10^8$	4,721.8	2,931.06	$6.2 \cdot 10^{-1}$

Table 8.12: Protected QuickSort on descending order data set.

8.1.6 Reflection

Besides recording performance measurements of the sorting algorithms we also implemented reflection in the sorting program.

Reflection allows a program to modify itself at runtime by dynamically assigning program code without knowing the names of classes, interfaces, fields, methods, etc. at compile time. This gives certain flexibility that allows a program to dynamically adapt its behavior without hard-coding it in advance.

Reflection has been included in the conducted tests, because it is an advanced feature present in the Java programming language and it is used by the TACTICOS technology stack. Therefore it is important to know if this feature still works after applying the code virtualization. The current TACTICOS technology stack would not be compatible with Solidshield if it does not support reflection. On top of that it is very practical to include reflection early-on because it helps to keep our test code elegant and flexible.

Although reflection causes performance overhead due to types that need to be resolved dynamically we consider this not to be a problem for our measurements because the reflection is only used to dynamically load the test objects defined in the configuration file and is therefore not part of the actual measuring performed on the sorting algorithms.

Before testing with reflection it was unclear if Solidshield would support this language feature properly. There are obfuscation technologies that change the names of classes, fields, methods, etc. Such protection tactics could potentially break reflection, because then the names on runtime could differ from the original names that were present at compile time. Virtualization could lead to similar problems if the original names and identifiers of the the protected virtualized code would be altered. Testing with our sorting program has shown however that Solidshield does support reflection. After decompiling and analyzing the protected versions of our sorting program we discovered why this is the case. The Solidshield tool leaves the interface-, class-, field-, and method-names untouched. Virtualization is performed on methods only. The method-body is virtualized and moved to the DVM where it is reconstructed by a publicly de-

fined method with the same name. This approach allows the control flow to be redirected to the DVM without requiring changes to the remaining unprotected code and let the DVM do its obfuscation and virtualization tricks. The program is unaware of the alterations made at bytecode level, because the method calls and return values remain the same, only the method-body implementation has been transformed and moved inside the DVM. When the virtualized method finishes its execution the control flow returns to the original program logic. Reflection is therefore supported by Solidshield in a transparent way and we do not foresee problems in that area for the TACTICOS technology stack.

To summarize our findings we found that Java reflection still works with Java code virtualization and we have shown that the encryption performance overhead is negligible but the code virtualization has huge impact on the performance.

8.2 TACTLESS Demo Application

The TACTLESS demo application has been tested and evaluated with the help of JMX. This allows instrumentation of the program to monitor and diagnose the protected versions. Metrics are gathered directly from the JVM. This can be done remotely allowing us to reduce potential non-determinism generated by the monitoring agent or other external sources, by running the TACTLESS demo application on a dedicated machine and connecting via the network to the JMX server. This approach of separating the actual simulation execution from other tasks should produce more reliable measurements. Besides monitoring the JMX is also used to manage the TACTLESS demo application. Most of the bundles include so called MBeans that we use to control and manipulate the TACTLESS demo application. This allows the demo application to be run headless, because it can be managed (remotely) with the MBeans making the GUI optional. It also allows dynamic manipulation of the bundles on runtime. This is ideal for experimenting and testing while simulating, because the initial configuration can be changed dynamically on any moment without stopping the program execution.

Profiling the TACTLESS demo application with realistic inputs shows how the dynamic behavior of a protected program is altered compared to its unprotected equivalent. This allows us to gather data and statistics such as the number of loaded classes, system properties, thread states, memory consumption, garbage collection, deadlock detection, etc. to analyze the demo application's behavior. We have selected several metrics to collect: CPU load, number of loaded classes, thread count, and memory usage. Memory can be divided even further into heap memory, non-heap memory, eden space, survivor memory, etc. but experimenting and comparing learned that this granularity does not really contribute to a better analysis of the performance. These metrics might however become important for a more detailed analysis to fine tune memory complexity for the 32-bit JVM where memory resources are restricted to a theoretical $2^{32} = 4.3$ gibibyte (GiB) maximum or < 4 gigabyte (GB). In reality the 4 GB limit is however closer to a maximum heap size of < 3.2 GB on linux hosts. This is an issue for TACTICOS because it currently still runs inside a 32-bit JVMs due to backwards compatibility with subsystems and legacy drivers.

Memory consumption is therefore critical and fine tuning could become necessary if virtualization is applied to the bigger and higher complexity code base of TACTICOS. Then the higher granularity of the Java memory resources metrics could help to identify problematic bottle necks and fine tune these. TACTICOS will migrate to a 64-bit JVM eventually but the additional metrics can be collected nevertheless when necessary.

The code that has been protected by the Solidshield tool can not be analyzed and debugged with the tools from the IDE. Preventing attackers from debugging and analyzing the software with an IDE, or any other tool for that matter, is obviously the point of applying the protection in the first place but it also hinders the developers when they develop the software because they also can not use these tools from their toolkit. Testing and evaluating of the TACTLESS demo application took therefore mainly place in an OSGi implementation as explained in Chapter 7.1. The runtimes gathered from the sorting algorithms in the sorting program and the TACTLESS demo application sorting bundles have been executed within the Karaf OSGi implementation. Unfortunately during the development phase we discovered that this approach can only be used to measure and monitor the performance. The managed beans implemented by our bundles were not accessible via JMX, because the bundles are executed inside Karaf containers, therefore we switched to a standalone Equinox runtime setup. The bundles running on Equinox did register with the JMX as intended and allow full control of the TACTLESS demo application via JMX.

8.2.1 Migrating Towards a Metrics Collecting Evaluation

Before jumping directly into the TACTLESS demo application results we will use this chapter to explain some of the effects observed in the collected data. We begin with metrics extracted from the sorting program and conclude by showing what the impact is of the OSGi environment on runtime performance.

Lets start by taking a look at the CPU load of the BubbleSort algorithm performed at an 100.000 element data set. Figure 8.1 shows that the monitoring starts halfway an iteration followed by five full iterations. The start of the graph is a little higher because the JMX agent connects to the sorting program in an other thread. The graph peeks at 12.5% because the BubbleSort implementation is single threaded and runs on a 4 core machine with support for Hyper-Threading Technology (HTT). This is an Intel adoptation of simultaneous multi-threading on their x86 architecture CPUs. Each physical HTT capable processor is divided into two virtual logical cores if the operating system supports it. The test machine therefore contains 8 logical cores, hence the 12,5% max the BubbleSort algorithm can achieve on merely one of the total of 8 logical cores. The JMX agent connecting runs in a separate thread on a different core making it possible to surpass the 12,5% limit. In fact there are 13 threads active during the execution of the sorting program, including 4 JMX related threads for monitoring and connecting clients, JVM threads for reference handling and signal dispatching, etc. This is important to remember because even a small single threaded program already has to compete with multiple threads from the JVM.

Looking at the memory consumption of the BubbleSort algorithm in Figure 8.2 shows a similar pattern as we saw with the CPU load. Every peak is an

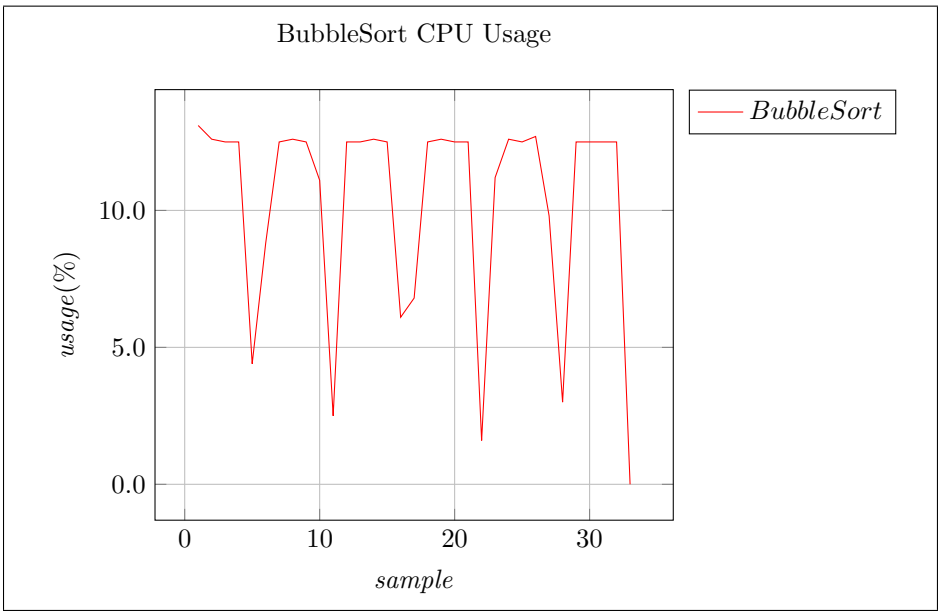

Figure 8.1: BubbleSort CPU Metrics - 100.000 elements.

iteration and the valleys are in fact caused by our testing approach when the sorted data set is removed from memory entirely.

When we combine the CPU load and the heap memory metrics we can see how time and space complexity relate to each other. Figure 8.3 shows how memory consumption of the BubbleSort algorithm fit together.

When we look at the combined graph of the BucketSort algorithm on 10.000.000 elements in Figure 8.4 we see how the monitoring started halfway a sequence of 30 iterations. The BucketSort algorithm is clearly more efficient because it produces less CPU load and consumes less memory while sorting a much larger data set then the BubbleSort example mentioned before. The valley in the CPU load graph followed by a peak in the heap usage is not an anomaly. This is caused due to the fact that the monitoring sample rate in this particular example was 4 seconds while the entire runtime of an unprotected BucketSort iteration on a 10.000.000 element data set is less then 2 seconds as demonstrated earlier in 8.3. The narrow peaks and valleys are therefore easily missed under these conditions.

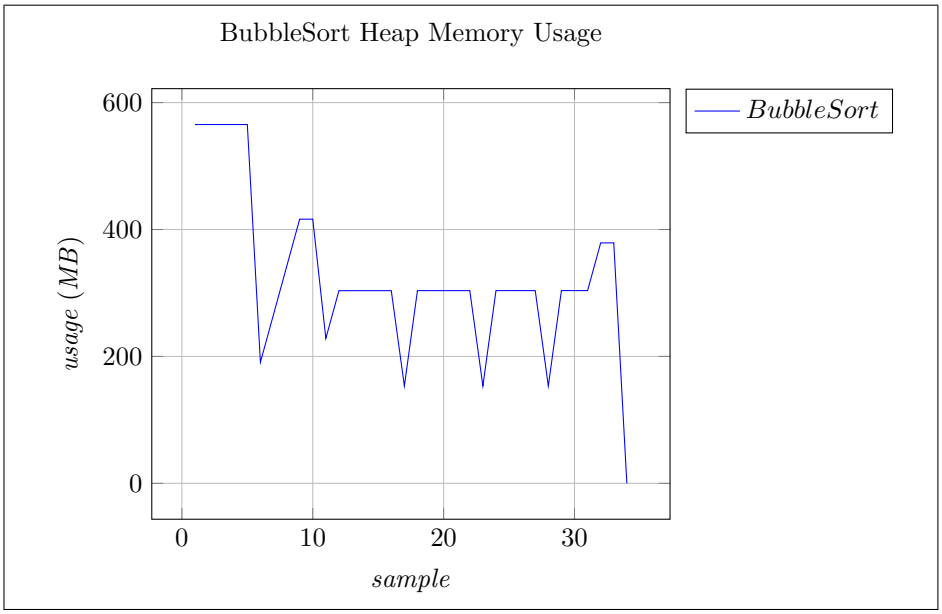

Figure 8.2: BubbleSort heap metrics - 100.000 elements.

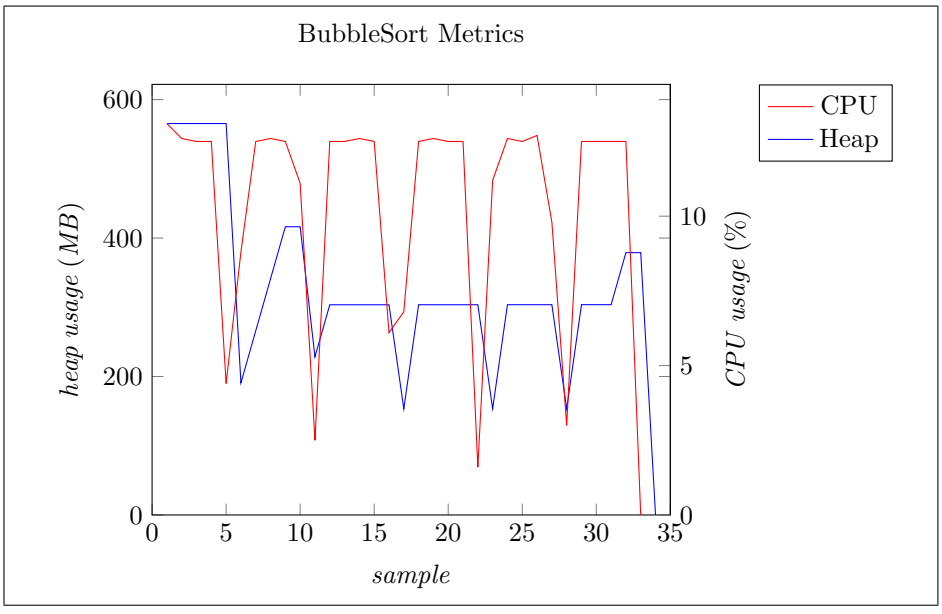

Figure 8.3: BubbleSort metrics combined - 100.000 elements.

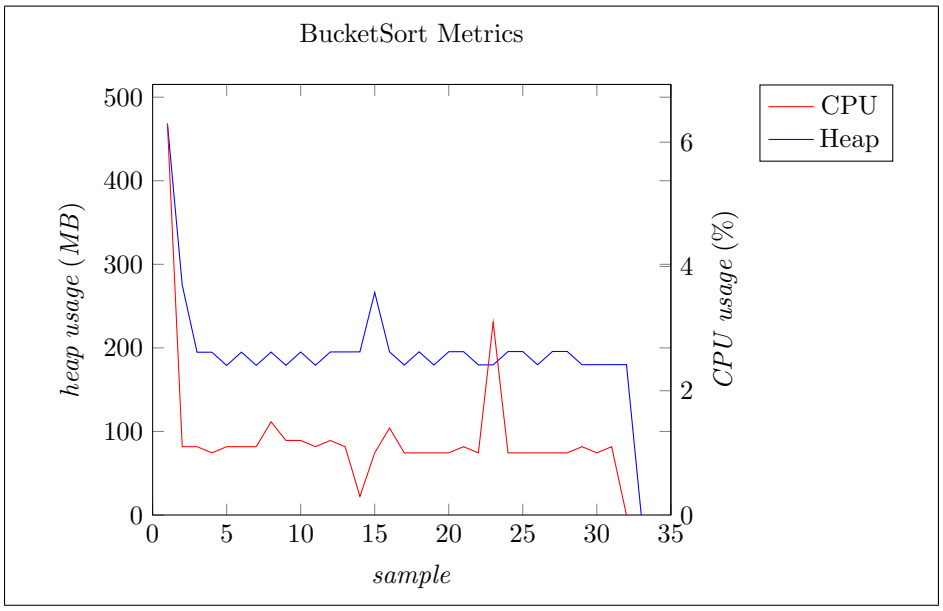

Figure 8.4: BucketSort metrics combined - 10.000.000 elements.

Now to make things really interesting we look at the threaded versions of the QuickSort algorithm. Figure 8.5 shows the QuickSortT implementation sorting 100.000.000 elements. The first 15 samples are dominated by heap activity but from the 17th sample and onward the heap remains stable around 1 GB.

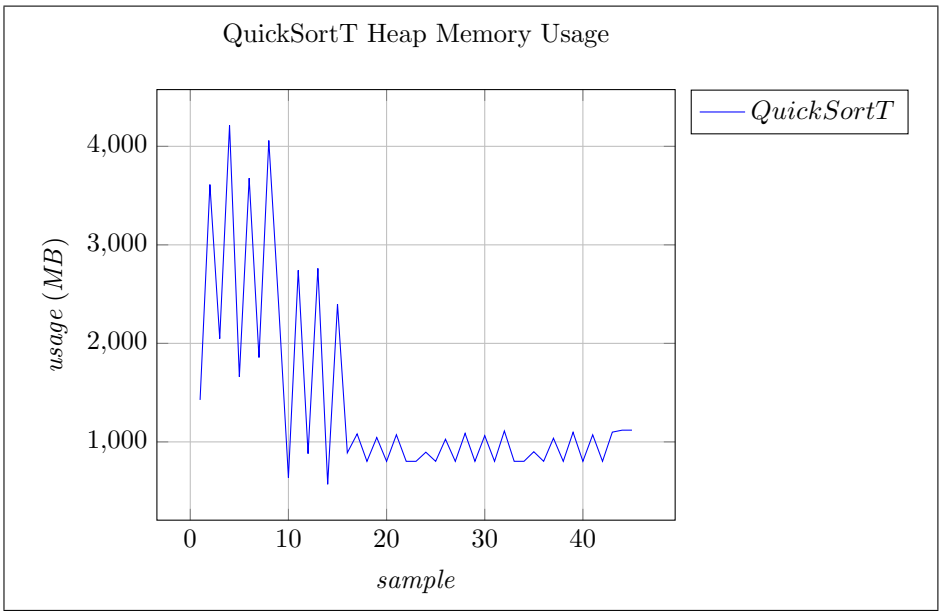

Figure 8.5: QuickSortT heap metrics - 100.000.000 elements.

Looking at the CPU load graph in figure 8.6 we show the opposite behavior. The first 15 samples show a stable load of 12,5% and onward the load increases up to over 40%.

What this effectively means is nicely illustrated by putting both graphs together in Figure 8.7. The big data set is first read into memory and this is handled by a single threaded file reader topping out at 12,5% load because it can only utilize one logical core. Once the data set has been loaded into memory it gets duplicated into a work data set to be sorted by the algorithm and then the algorithm is set to work. The multi-threaded version utilizes more logical cores and therefore the CPU load can exceed the 12,5% limit. This ability to employ more then one logical core leads to faster processing, because the 12,5% bottleneck single core limitation is removed and therefore it can sort more elements then the single threaded QuickSort in the same time and complete its task sooner.

As one might expect the QuickSortTT implementation shows similar behavior in Figure 8.8. The unprotected threaded versions of QuickSort were evenly matched as discussed in Chapter 8.1.4, with a slight advantage to QuickSortT for data sets up to 100.000 elements. From data sets of 1.000.000 elements and beyond QuickSortTT performed better.

Putting the CPU load metrics of both algorithms together in one graph, illustrated by Figure 8.9, confirms this but it does show that QuickSortT has higher CPU load peaks. During the sorting program testing we speculated that the QuickSortT algorithm was a bit more aggressive resulting in the performance loss for the bigger data sets and now with these metrics it becomes apparent

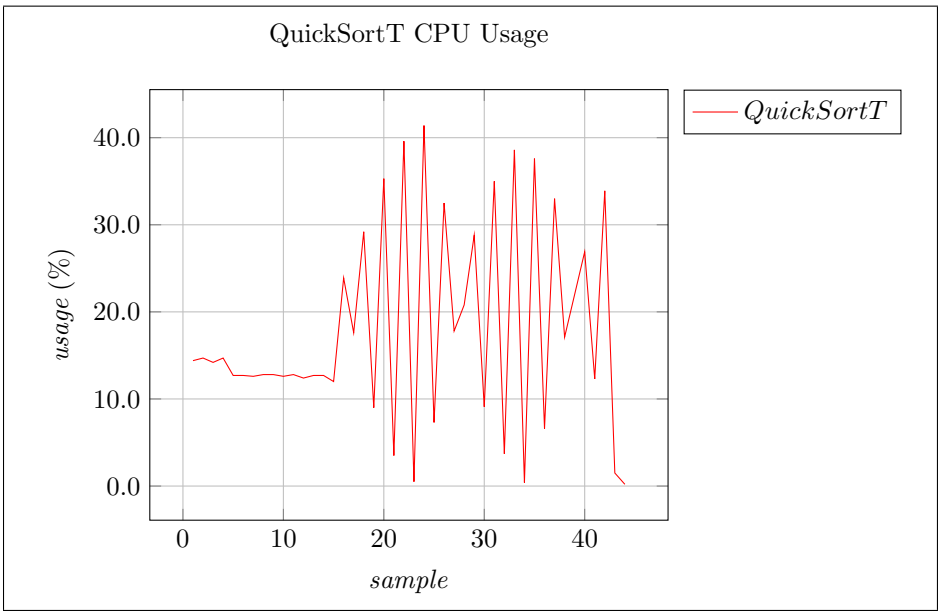

Figure 8.6: QuickSortT CPU metrics - 100.000.000 elements.

that this effect plays a role and it is indeed measurable.

Furthermore it is important to mention that the thread count for QuickSortT and QuickSortTT stabilized around 19 active threads during execution with incidental peaks to 21 active threads. The non-threaded algorithms had 13 active threads. This makes perfect sense because that are the 7 additional utilized logical cores on the 8 logical cores machine as opposed to the single logical core used by the single threaded algorithms.

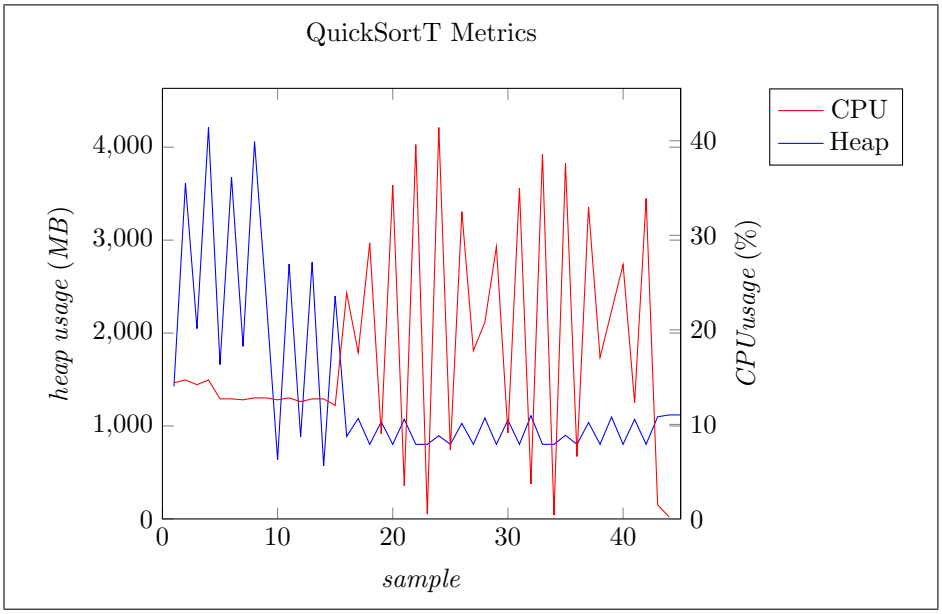

Figure 8.7: QuickSortT combined metrics - 100.000.000 elements.

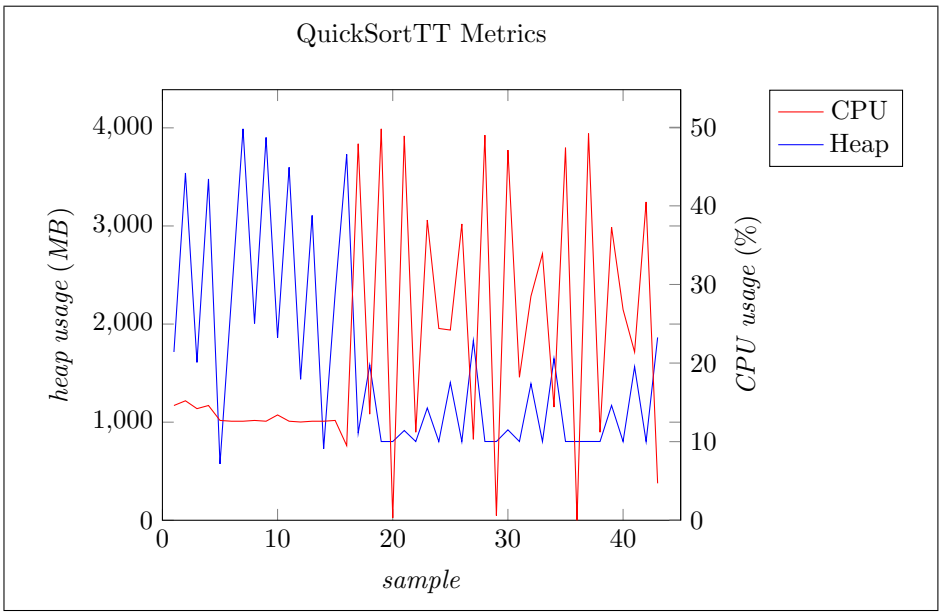

Figure 8.8: QuickSortTT combined metrics. - 100.000.000 elements

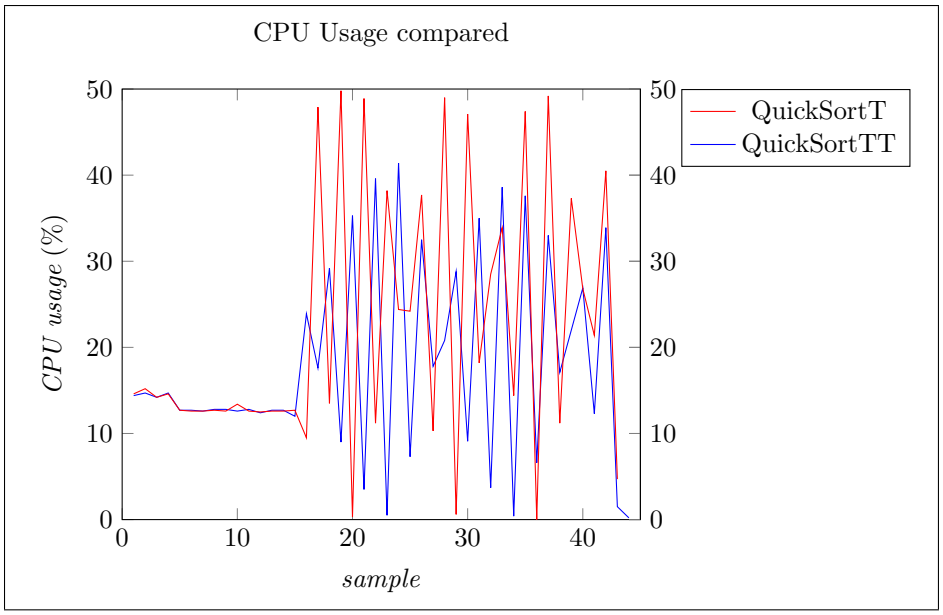

Figure 8.9: QuickSortT vs QuickSortTT CPU usage - 100.000.000 elements.

Now it is time to look at some of the metrics extracted from test runs with protected versions of our sorting algorithms. We start by looking at a protected version of BucketSort in Figure 8.10. Where we saw that the unprotected reference version was very efficient in processing the larger 10.000.000 element data set with only a few percent CPU load and roughly 200 megabyte (MB) heap memory on average, the virtualized version performs significantly worse on the same input set. The heap consumption has increased by a factor 10 almost up to 2 GB and the single threaded CPU load tops out at 12,5% using up the entire logical core. Based on the metrics gathered we can deduce that the DVM does not create additional threads, because the CPU load tops out at 12,5% and the thread count stays the same. This would imply that even though there is control flow obfuscation applied it does not spawn new threads and after the transfer of control to the DVM the flow is executed sequentially just like the original code.

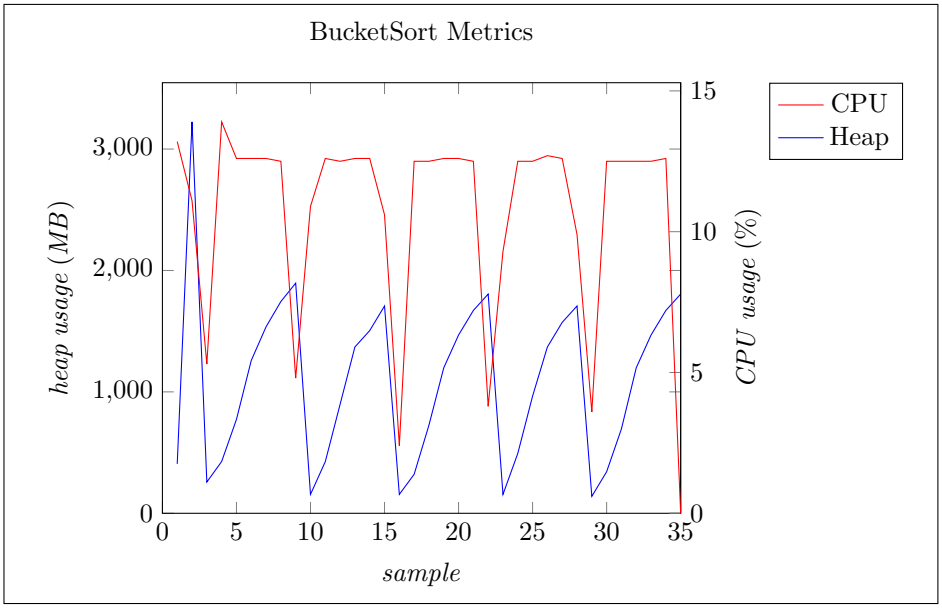

Figure 8.10: Protected BucketSort metrics combined - 10.000.000 elements.

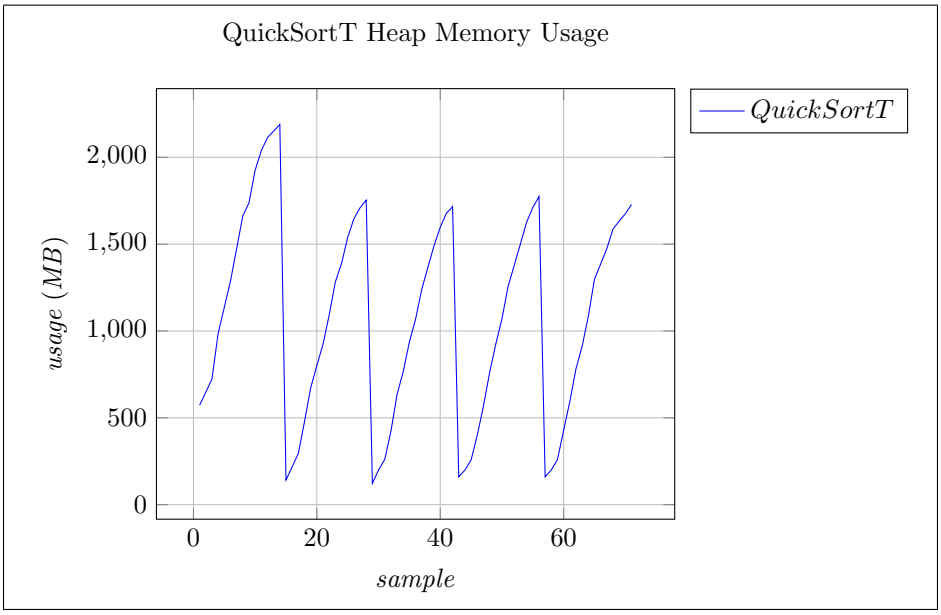

Figure 8.11: Protected QuickSortT heap metrics - 1.000.000 elements.

The last important observation we would like to address here is the fact that the protected version of QuickSortT performed much worse than the protected version of QuickSortTT while we saw in Figure 8.9 that the unprotected reference versions behaved very similar. Looking at the QuickSortT metrics in 8.12 and the QuickSortTT metrics in 8.15 show that both algorithms utilize up to slightly less then 80% of the CPU resources. Memory wise the QuickSortTT appears to perform better on average, but when putting the CPU load metrics together into 8.16 the problem becomes abundantly clear. The QuickSortT takes much longer to process a single iteration. In Chapter 8.1.4 we speculated that the more aggressive threading strategy might cause this behavior. To rule out this effect QuickSortT has been slightly adapted to not exceed the maximum available logical cores. The thread-count and number of active threads are now the same for both implementations. This potential cause has now been ruled out by the threading metrics collected during these measurements. The problem however remains. One iteration claims the CPU cores much longer and the load scales down at a slower rate but according to the metrics that effect can not be explained by a higher thread count. When we compare the memory consumption in Figure 8.17 however then it appears that memory usage is higher for QuickSortT. This is odd because both algorithms follow the same quicksort strategy, the only difference is the threading implementation. The threaded implementation takes advantage of parallelization and adds some overhead, but knowing that the space complexity and the input data set are the same for both implementations one would expect similar memory consumption from a deterministic algorithm. Threading introduces some non-determinism causing slight variations in execution but the operations and swaps should be the same on the isolated partitions regardless. Time and space complexity are very similar for the unprotected versions as we saw in Figure 8.8 and their respective heap consumption graphs shown in Figure 8.11 and 8.13. The unprotected QuickSortT

actually performed a little better with an steady average of 1 GB memory usage while QuickSortTT had slightly more variation with peaks up to over 1.5 GB. The protected QuickSortT however uses up to double that amount while the QuickSortTT does not require such a memory boost.

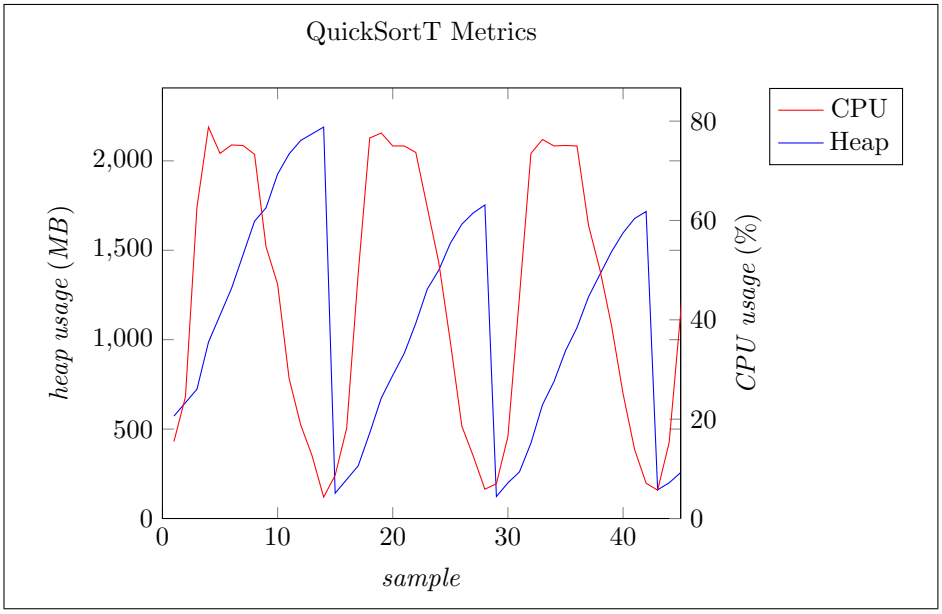

Figure 8.12: Protected QuickSortT combined metrics - 1.000.000 elements.

During testing of the sorting program we ran into a thread synchronization bug. This caused problems such as race conditions resulting in wrong sorting results and even exceptions and lock ups as mentioned in 8.1.4. The problem has been solved by the Soldshield developers but the behavior observed in QuickSortT is obviously different to QuickSortTT after the Solidshield protection has been applied. The results may now be correct but the performance is much worse. We ruled out the number of active threads as a possible cause and do not know for sure what causes QuickSortT to be less efficiently protected by Solidshield but the effect is undeniable present. Some constructs are handled apparently more efficient then others by the DVM.

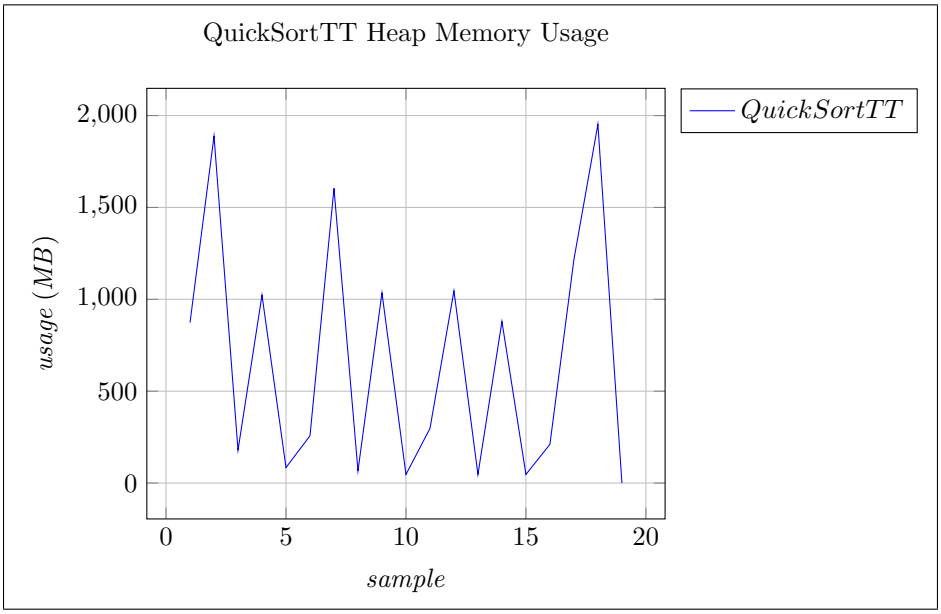

Figure 8.13: Protected QuickSortTT heap metrics - 1.000.000 elements.

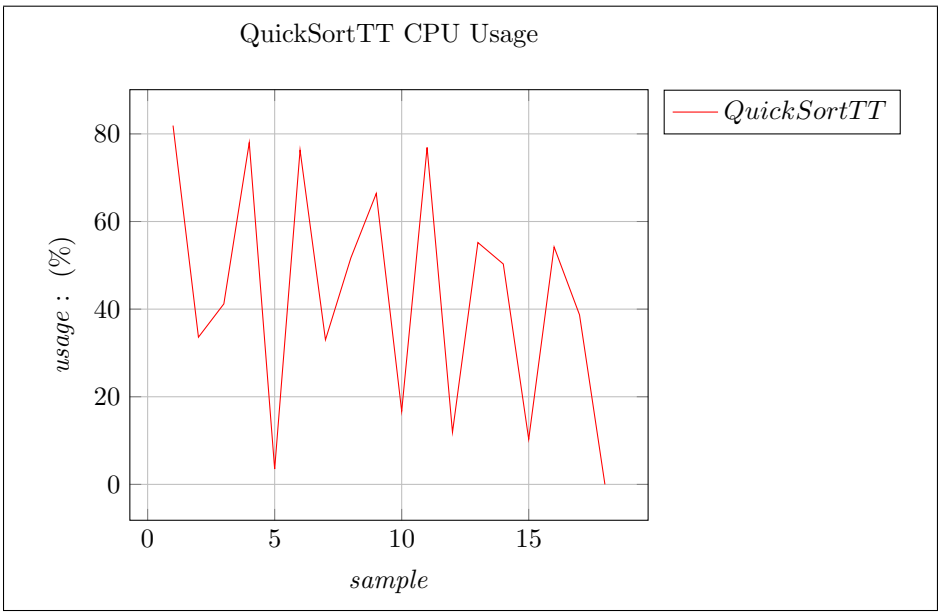

Figure 8.14: Protected QuickSortTT CPU metrics - 1.000.000 elements.

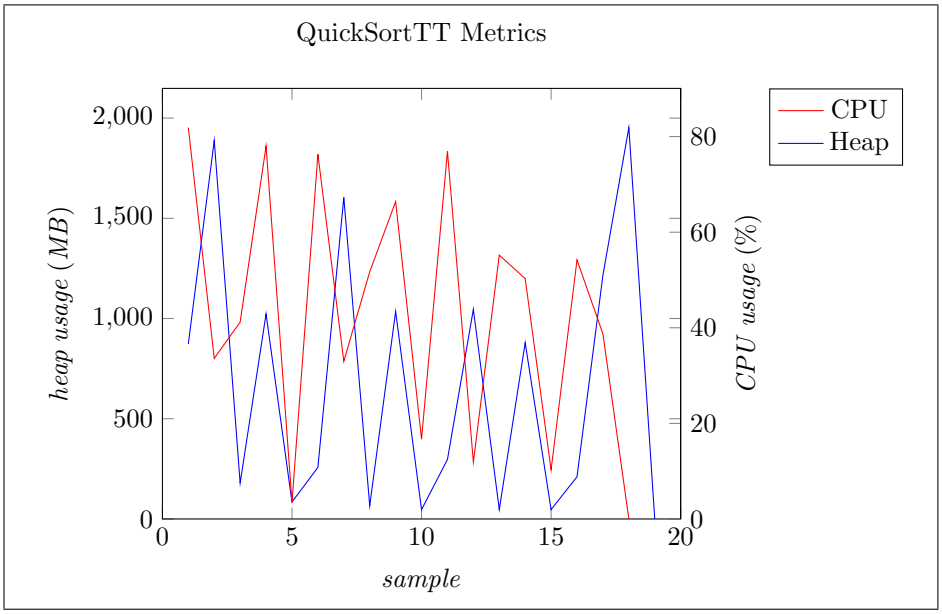

Figure 8.15: Protected QuickSortTT combined metrics - 1.000.000 elements.

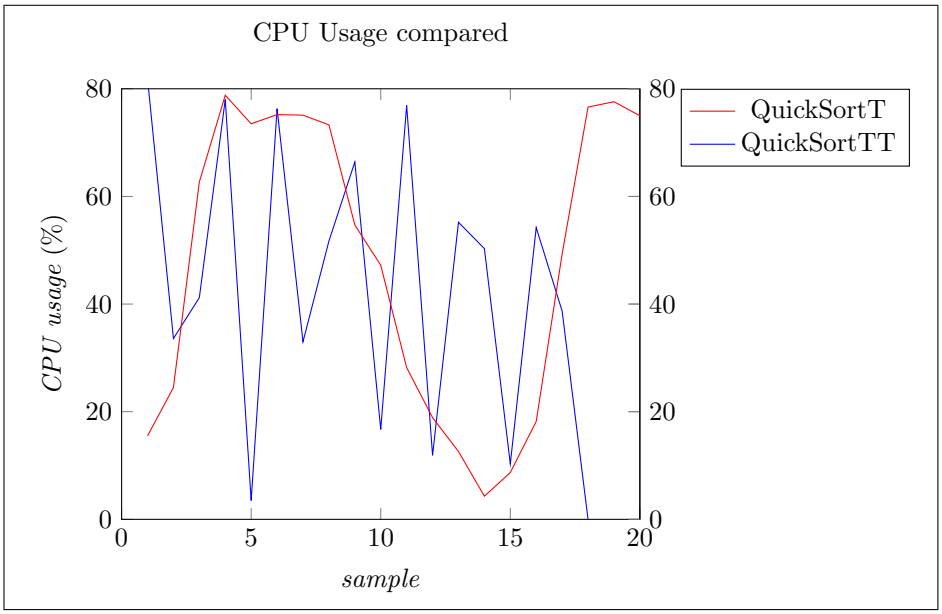

Figure 8.16: Protected QuickSortT vs QuickSortTT CPU usage - 1.000.000 elements.

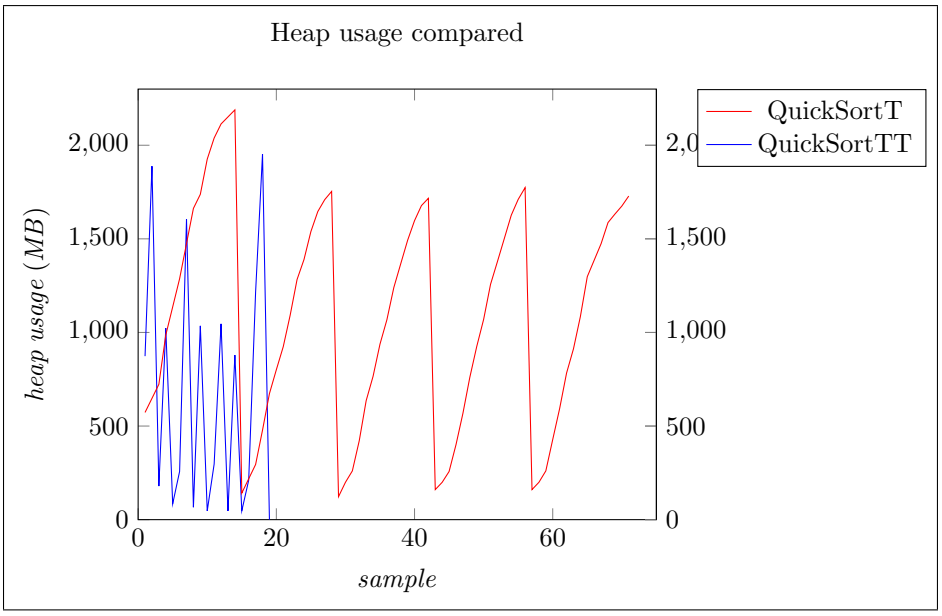

Figure 8.17: Protected QuickSortT vs QuickSortTT heap memory usage - 1.000.000 elements.

8.2.2 OSGi Environment Impact

The sorting algorithms have been extensively tested in a non-OSGi environment and we use that to study the influence of OSGi on the Solidshield virtualization. They have been re-implemented as OSGi bundles and as such used in the TACTLESS demo application to get a good grasp on the impact of virtualized bundles inside an OSGi environment.

First we establish the impact of OSGi on the unprotected algorithms by comparing the sorting bundles runtimes to the original sorting algorithm runtime baselines. The results are presented in tables 8.13 to 8.17.

The tables presented here have been constructed from measurements performed inside an OSGi environment. More detailed runtime measurements of the unprotected reference bundle and the protected bundle can be found in the Appendix F.1.

There are obviously some differences between the sorting algorithms and the sorting bundles. Every sorting algorithm has been redesigned and re-factored into an OSGi bundle. Every sorting bundle offers an algorithm as a service and adds therefore some additional overhead from the OSGi runtime and component framework. This is visible when looking at collected metrics, i.e. the thread count and number of loaded classes are higher. The runtimes are however surprisingly similar. Figure 8.13 shows that factor lies around 1 for the BubbleSort bundle compared to the original BubbleSort algorithm runtime. The small input sets results should be ignored because the runtimes are too short for reliable measurements as indicated by the calculated standard deviations in Appendix F.1. The same is true for the other sorting bundles as shown in Figure 8.14, 8.15, 8.16 and 8.17.

elements	*baseline* (*s*)	*OSGi* (*s*)	*factor*
$1 \cdot 10^1$	$5.8 \cdot 10^{-5}$	$6{\cdot}10^{-5}$	1.03
$1 \cdot 10^2$	$3.72 \cdot 10^{-4}$	$1.21 \cdot 10^{-4}$	0.32
$1 \cdot 10^3$	$9.64 \cdot 10^{-4}$	$9.49 \cdot 10^{-4}$	0.98
$1 \cdot 10^4$	0.11	0.11	1.03
$1 \cdot 10^5$	12.67	12.78	1.01
$1 \cdot 10^6$	1,252.2	1,264.04	1.01

Table 8.13: BubbleSort bundle performance.

The performance of the bundles inside the OSGi runtime is quite good judging by the calculated factors that approach 1 for the bigger data sets. Based on these results we can conclude that the added overhead from the migration to an OSGi environment is not significant.

Protecting the sorting bundles by virtualization however shows that this is not the case for virualization inside an OSGi environment. Tables 8.18 to 8.21 show the sorting algorithm bundle reference runtime baseline, the runtime of the protected version and the performance factor relation between these two runtimes. The *original* factor represents the performance factor of the non-OSGi sorting algorithms discussed in Chapter 8.1.

BubbleSort shown in Table 8.18 says it all. The performance factor already increased a lot on the original non-OSGi BubbleSort version. Virtualized it

elements	*baseline* (*s*)	*OSGi* (*s*)	*factor*
$1 \cdot 10^1$	$6.9 \cdot 10^{-5}$	$5.8 \cdot 10^{-5}$	$8.4 \cdot 10^{-1}$
$1 \cdot 10^2$	$7.25 \cdot 10^{-5}$	$5.5 \cdot 10^{-5}$	$7.6 \cdot 10^{-1}$
$1 \cdot 10^3$	$7.2 \cdot 10^{-5}$	$7 \cdot 10^{-5}$	$9.7 \cdot 10^{-1}$
$1 \cdot 10^4$	$1.45 \cdot 10^{-4}$	$1.28 \cdot 10^{-4}$	$8.8 \cdot 10^{-1}$
$1 \cdot 10^5$	$1 \cdot 10^{-3}$	$9.87 \cdot 10^{-4}$	$9.8 \cdot 10^{-1}$
$1 \cdot 10^6$	$2.97 \cdot 10^{-3}$	$2.92 \cdot 10^{-3}$	$9.9 \cdot 10^{-1}$
$1 \cdot 10^7$	0.16	0.15	$9.4 \cdot 10^{-1}$
$1 \cdot 10^8$	1.99	1.88	$9.5 \cdot 10^{-1}$

Table 8.14: BucketSort bundle performance.

elements	*baseline* (*s*)	*OSGi* (*s*)	*factor*
$1 \cdot 10^1$	$7.74 \cdot 10^{-5}$	$7 \cdot 10^{-5}$	$9 \cdot 10^{-1}$
$1 \cdot 10^2$	$7.69 \cdot 10^{-5}$	$5.8 \cdot 10^{-5}$	$7.5 \cdot 10^{-1}$
$1 \cdot 10^3$	$2.35 \cdot 10^{-4}$	$1.12 \cdot 10^{-4}$	$4.8 \cdot 10^{-1}$
$1 \cdot 10^4$	$8.31 \cdot 10^{-4}$	$7.86 \cdot 10^{-4}$	$9.5 \cdot 10^{-1}$
$1 \cdot 10^5$	$8.88 \cdot 10^{-3}$	$8.75 \cdot 10^{-3}$	$9.9 \cdot 10^{-1}$
$1 \cdot 10^6$	$7.51 \cdot 10^{-2}$	$7.47 \cdot 10^{-2}$	$1 \cdot 10^0$
$1 \cdot 10^7$	1.14	1.12	$9.8 \cdot 10^{-1}$
$1 \cdot 10^8$	12.98	12.79	$9.9 \cdot 10^{-1}$

Table 8.15: QuickSort bundle performance.

performed up to 732 times worse on 1.000 elements and 640 times worse for 10.000 elements. The performance for the OSGi version of BubbleSort has deteriorated hugely. The unprotected bundle managed to sort 1.000 elements in 0.11 seconds but virtualized it took almost 286 seconds. That is a massive increase and leads up to a factor of 2.977 on 1.000 elements and it performs 2.494 times worse for 10.000 elements. Sorting the 10.000 element set had to be aborted due to the time it would have required to finish one iteration.

BucketSort in Table 8.19 shows a similar outcome albeit a bit more modest. The peak for the 1.000.000 data set should be ignored as explained in Chapter 8.1.2 because it does not only reflect the performance deterioration but also an optimization by the JVM on the data set that only happens for the 1.000.000 element set. Looking at the other data sets then the performance deterioration is still significant. The non OSGi BucketSort algorithm performed between 90 and 172 times worse after virtualization. The OSGi BucketSort bundle however performed from 440 up to almost 832 times worse.

QuickSort presented in Table 8.20 also suffers from the same deterioration. The algorithm performed roughly a factor 100 worse for the bigger data sets after virtualization was applied. In an OSGi environment however the performance worsens roughly up to a factor of 380 when the 1.000.000 data set is ignored.

The threaded QuickSortTT implementations shows an interesting result in Table 8.21. Comparing the OSGi version to the original hows pretty decent results after virtualization. As usual the smaller data sets should be ignored due to the short runtimes and relative large standard deviation. Looking at

elements	*baseline* (s)	*OSGi* (s)	*factor*
$1 \cdot 10^1$	$1.45 \cdot 10^{-4}$	$2.13 \cdot 10^{-4}$	1.5
$1 \cdot 10^2$	$2.12 \cdot 10^{-4}$	$2.28 \cdot 10^{-4}$	1.1
$1 \cdot 10^3$	$1.68 \cdot 10^{-4}$	$1.86 \cdot 10^{-4}$	1.1
$1 \cdot 10^4$	$3.6 \cdot 10^{-4}$	$3.65 \cdot 10^{-4}$	1
$1 \cdot 10^5$	$2.81 \cdot 10^{-3}$	$2.7 \cdot 10^{-3}$	$9.6 \cdot 10^{-1}$
$1 \cdot 10^6$	$3.59 \cdot 10^{-2}$	$3.67 \cdot 10^{-2}$	1
$1 \cdot 10^7$	0.27	0.27	$9.8 \cdot 10^{-1}$
$1 \cdot 10^8$	3.06	3.03	$9.9 \cdot 10^{-1}$

Table 8.16: QuickSortT bundle performance.

elements	*baseline* (s)	*OSGi* (s)	*factor*
$1 \cdot 10^1$	$2.32 \cdot 10^{-4}$	$3.33 \cdot 10^{-4}$	1.4
$1 \cdot 10^2$	$1.87 \cdot 10^{-4}$	$2.74 \cdot 10^{-4}$	1.5
$1 \cdot 10^3$	$2.32 \cdot 10^{-4}$	$2.71 \cdot 10^{-4}$	1.2
$1 \cdot 10^4$	$5 \cdot 10^{-4}$	$4.62 \cdot 10^{-4}$	$9.2 \cdot 10^{-1}$
$1 \cdot 10^5$	$3.24 \cdot 10^{-3}$	$3.13 \cdot 10^{-3}$	$9.7 \cdot 10^{-1}$
$1 \cdot 10^6$	$2.76 \cdot 10^{-2}$	$2.8 \cdot 10^{-2}$	1
$1 \cdot 10^7$	0.22	0.22	1
$1 \cdot 10^8$	2.33	2.25	$9.7 \cdot 10^{-1}$

Table 8.17: QuickSortTT bundle performance.

the bigger data sets it performs worse but it appears to be affected less heavily compared to the non-threading algorithms.

We established that migrating the sorting algorithms to an OSGi environment did not affect the performance of the unprotected algorithms much. Applying virtualization however did have significant consequences for the runtimes. Somehow the virtualized algorithm performance is thus affected more when running in an OSGi environment. There is actually a logical explanation for this behavior because there is an important difference between the protected sorting algorithms and the protected OSGi sorting bundles. The sorting algorithms are protected by one DVM because they are part of one program that is distributed in a single JAR file. As explained in Chapter 5 the protection is applied to methods on the level of JAR files. Every OSGi bundle is distributed as a JAR file. Protecting one or more methods in a bundle requires a DVM to be added in that bundle. Every protected bundle comes therefore with its own DVM and that obviously introduces a lot of overhead.

	Runtimes		Factor	
elements	*baseline* (*s*)	*protected* (*s*)	*OSGi*	*original*
$1 \cdot 10^1$	$6{\cdot}10^{-5}$	$3.95 \cdot 10^{-4}$	6.6	3.6
$1 \cdot 10^2$	$1.21 \cdot 10^{-4}$	$2.61 \cdot 10^{-2}$	215.8	19.5
$1 \cdot 10^3$	$9.49 \cdot 10^{-4}$	2.83	2,977.5	732.6
$1 \cdot 10^4$	0.11	285.65	2,494.4	640.5

Table 8.18: OSGi impact on virtualized BubbleSort.

	Runtimes		Factor	
elements	*baseline* (*s*)	*protected* (*s*)	*OSGi*	*original*
$1 \cdot 10^1$	$5.8 \cdot 10^{-5}$	$2.72 \cdot 10^{-4}$	4.7	2.8
$1 \cdot 10^2$	$5.5 \cdot 10^{-5}$	$9.65 \cdot 10^{-4}$	17.5	6
$1 \cdot 10^3$	$7{\cdot}10^{-5}$	$8.34 \cdot 10^{-3}$	119.2	29.5
$1 \cdot 10^4$	$1.28 \cdot 10^{-4}$	$8.22 \cdot 10^{-2}$	642.1	119.3
$1 \cdot 10^5$	$9.87 \cdot 10^{-4}$	0.82	831.7	172.1
$1 \cdot 10^6$	$2.92 \cdot 10^{-3}$	8.19	2,801.1	578.9
$1 \cdot 10^7$	0.15	82.78	545.2	110.9
$1 \cdot 10^8$	1.88	828.87	439.9	90.4

Table 8.19: OSGi impact on virtualized BucketSort.

	Runtimes		Factor	
elements	*baseline* (*s*)	*protected* (*s*)	*OSGi*	*original*
$1 \cdot 10^1$	$7{\cdot}10^{-5}$	$2.85 \cdot 10^{-4}$	4.1	7.1
$1 \cdot 10^2$	$5.8 \cdot 10^{-5}$	$1.93 \cdot 10^{-3}$	33.3	11.9
$1 \cdot 10^3$	$1.12 \cdot 10^{-4}$	$2.25 \cdot 10^{-2}$	201.3	44.1
$1 \cdot 10^4$	$7.86 \cdot 10^{-4}$	0.27	347.7	88.9
$1 \cdot 10^5$	$8.75 \cdot 10^{-3}$	3.32	379.1	101.6
$1 \cdot 10^6$	$7.47 \cdot 10^{-2}$	35.8	479	129.4
$1 \cdot 10^7$	1.12	428.58	382.1	103
$1 \cdot 10^8$	12.79	4,803.44	375.5	102

Table 8.20: OSGi impact on virtualized QuickSort.

	Runtimes		Factor	
elements	*baseline* (*s*)	*protected* (*s*)	*OSGi*	*original*
$1 \cdot 10^1$	$2.32 \cdot 10^{-4}$	$5.78 \cdot 10^{-4}$	2.5	8.6
$1 \cdot 10^2$	$1.87 \cdot 10^{-4}$	$1.03 \cdot 10^{-3}$	5.5	28.6
$1 \cdot 10^3$	$2.32 \cdot 10^{-4}$	$6.68 \cdot 10^{-3}$	28.8	55.1
$1 \cdot 10^4$	$5{\cdot}10^{-4}$	$6.12 \cdot 10^{-2}$	122.3	137.1
$1 \cdot 10^5$	$3.24 \cdot 10^{-3}$	0.6	186.6	175.9
$1 \cdot 10^6$	$2.76 \cdot 10^{-2}$	8.89	322.4	255.4
$1 \cdot 10^7$	0.22	60.4	272.6	238.9
$1 \cdot 10^8$	2.33	606.85	260.6	228

Table 8.21: OSGi impact on virtualized QuickSortTT.

8.2.3 Netty Bundle

Experiments with the sorting algorithms have shown that there are significant performance penalties for virtualized code. It is therefore not difficult to end up with protected methods that are rendered useless due to the performance loss. The question is however if it is possible to apply code vitualization at acceptable cost. This question will be answered by looking at the Netty bundle implemented in the TACTLESS demo application. As explained in Chapter 7, TACTLESS is developed as a demo application to roughly approximate the TACTICOS behavior.

Based on our experiences from testing with the sorting algorithms and bundles we optimize the code that hast to be protected by virtualization. One of the lessons learned was that certain operations are more expensive then others. Expensive operations are for example (nested) loops, string concatenation and function calls. Optimization to reduce the virtualization costs could be achieved by addressing these problems. Minimizing the use of getters and setters in virtualized code to reduces the number of function calls helps. Moving loops outside protected method-bodies is very beneficial and string concatenation can be circumvented in situations where a Java Stringbuffer suffices.

The example discussed here shows the effects of virtualization on an update method for the ADS-B protocol. This update method applies updates to the tracks altitude, speed, etc. naively with random updates. The logic for computing is therefore relatively simple and could be implemented without the use of loops. There are a few function calls that could be even further optimized.

Figure 8.18 shows the results of virtualizing such an optimized method in an realistic environment. At first glance the roundtrip times of the tracks in the system appear to be affected minimally. The roundtrip time of the protected bundle P is on average only 0.01 seconds slower then the unprotected reference bundle R for 1.000 tracks at an update rate of 1 Hz (1 update per second). This leads to a practical performance factor that lies around 1 for the tracks.

Looking at the execution time of the update method shown in Figure 8.19 however shows that the protected bundle is significantly slower then the unprotected reference version. Table 8.22 shows a comparison of the roundtrip times and the update method execution times between the unprotected reference R bundle and the protected bundle P. The calculated performance for 1000 tracks with an 1 Hz update rate is roughly a factor 75 worse. This shows that the virtualization protection is still relatively expensive but due to the fact that it only makes a small contribution to the total roundtrip time of the tracks in the system its influence is minimal on the overall bundle performance.

	Roundtrip			Update		
$Tracks$	$R\,(s)$	$P\,(s)$	Fac	$R\,(s)$	$P\,(s)$	Fac
10	$8.72 \cdot 10^{-4}$	$8.49 \cdot 10^{-4}$	0.97	$8.24 \cdot 10^{-7}$	$1.55 \cdot 10^{-5}$	19
100	$1.03 \cdot 10^{-2}$	$1.08 \cdot 10^{-2}$	1.04	$4.7 \cdot 10^{-7}$	$1.36 \cdot 10^{-5}$	29
1,000	0.1	0.11	1.08	$4.4 \cdot 10^{-7}$	$3.31 \cdot 10^{-5}$	75

Table 8.22: ADS-B Netty networking bundle comparison at 1 Hz.

It is important to observe that Table 8.23 and Table 8.24 show a large standard deviation for the roundtrip times with only 10 tracks. The short roundtrip

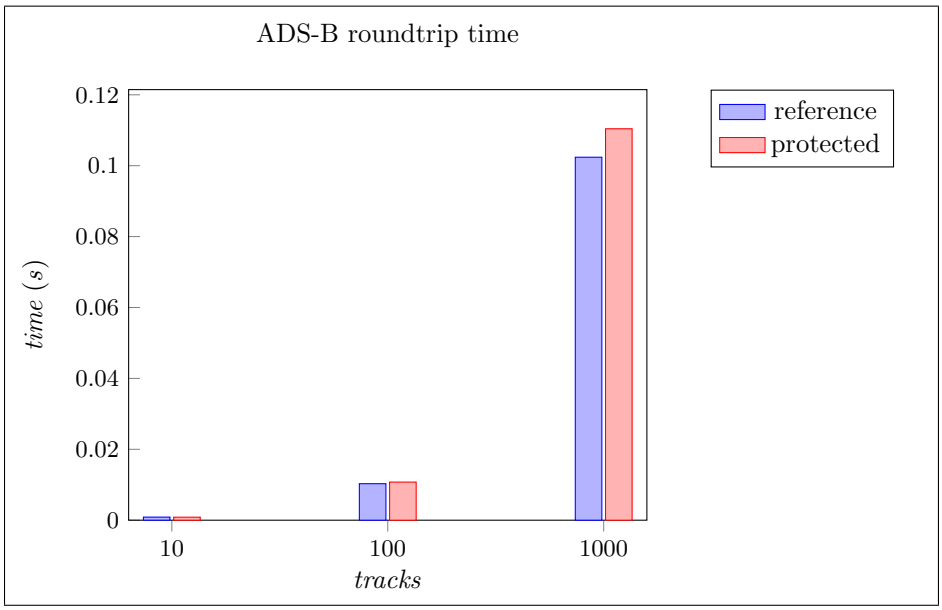

Figure 8.18: ADS-B protocol roundtrip time in Netty networking bundle.

time has relatively high variation making the average an unreliable indicator. Conclusions can not be drawn from these measurements and they should therefore be ignored. The measurements with with 100 tracks also suffer from slightly from this problem but to a lesser degree. Results from the measurements with 1.000 tracks are more accurate.

tracks	*roundtrip* (s)	σ	*update execution* (s)
10	$8.72 \cdot 10^{-4}$	$1.4 \cdot 10^{-3}$	$8.24 \cdot 10^{-7}$
100	$1.03 \cdot 10^{-2}$	$8.07 \cdot 10^{-3}$	$4.7 \cdot 10^{-7}$
1,000	0.1	$3.83 \cdot 10^{-2}$	$4.4 \cdot 10^{-7}$

Table 8.23: ADS-B Netty networking bundle measurements at 1 Hz.

tracks	*roundtrip* (s)	σ	*update execution* (s)
10	$8.49 \cdot 10^{-4}$	$1.12 \cdot 10^{-3}$	$1.55 \cdot 10^{-5}$
100	$1.08 \cdot 10^{-2}$	$7.84 \cdot 10^{-3}$	$1.36 \cdot 10^{-5}$
1,000	0.11	$3.63 \cdot 10^{-2}$	$3.31 \cdot 10^{-5}$

Table 8.24: Protected ADS-B Netty networking bundle measurements at 1 Hz.

The update frequency of 1 Hz used here is a realistic value. At the chosen update rate it is possible to increase the number of tracks even further up to several thousands before the roundtrip time becomes too large to complete within the window of 1 Hz. Decreasing the update frequency gives no problem. Then the bundle has more time to complete the updates and send/receive the tracks. Increasing the update frequency however is limited by the roundtrip

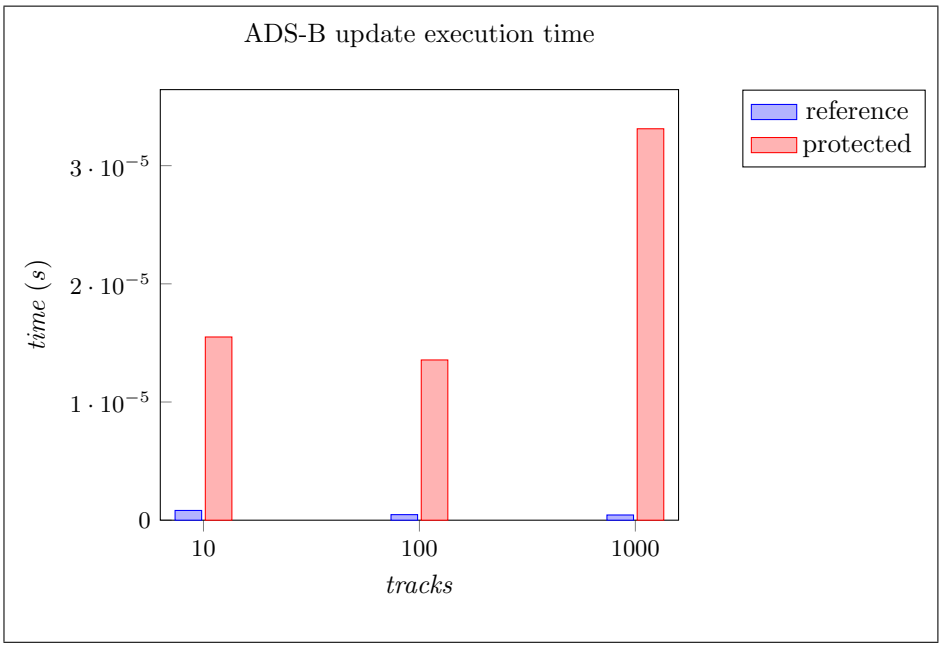

Figure 8.19: ADS-B protocol updates in Netty networking bundle.

time. When the roundtrip time surpasses the available time between the updates the system gets clogged up. With high refresh rates this can spiral out of control quickly leading to a unresponsive bundle. In our example with 1.000 tracks the roundtrip time was 0.1 seconds. The theoretical update frequency could therefore be 10 Hz, thus updating the tracks every 0,1 second. The protected bundle however would not be able to complete these updates within the 0,1 second window, because the roundtrip time is 0,11 second. This means it is 0,01 seconds too slow to meet the criteria.

The important observation to be made here is that acceptable performance depends on the context. An protected method with a short runtime the performance loss can be barely measurable but when it is used intensively and recurring it might exceed the acceptable range of a prior established performance zone, while an method with many expensive operations might suffers from a huge performance penalty and yet be perfectly acceptable when it is not time sensitive or not occurring very often.

Chapter 9

Discussion

In Chapter 3 we motivate the methodology applied to collect meaningful measurements in a statistically robust evaluation approach. This is especially important for Java program evaluation because the JVM introduces additional non-determinism on top of the operating system and hardware related non-determinism already present in a system. This is a known problem and we tried to address this in the methodology but during the benchmark tests it became evident that the variance for operations with short runtimes was relatively big. This can be observed in Chapter 8 by looking at the sorting algorithm runtimes for the small data sets. These measurements are less accurate, because the variability is relatively bigger with these short runtimes. Fortunately the statistically rigorous approach produces more accurate average results and the calculated standard deviation indicates when measurements are unreliable. The overall trends regarding performance and scalability of the protection under investigation are therefore considered to be reliable.

In Chapter 4 we introduced several sorting algorithms and a sorting program. We experimented with several disassemblers and decompilers to find out how much code could be recovered. Decompiling the sorting program resulted in source code nearly identical to the original source code including the original variable and method names. An interesting detail was that the decompiled program revealed a compilation optimization applied to the bytecode. Several post increments had been replaced by pre increments, which is slightly more efficient because the former has to make an additional copy of the original variable before it is incremented whereas the latter increments the variable directly. When the compiler detects that the copy is not used in any expression it can safely change a post increment to a pre increment. The same is obviously true for decrements.

In Chapter 5 we introduced the virtualization tool. By dissecting the Solidshield protection trough disassembling and decompiling protected sorting algorithms we managed to get a good grasp on how the protection works. Decompiling is not trivial because the protected code does contain some obfuscation techniques that make it difficult for decompilers to generate correct Java source code. With the right tools, some creativity and determination decent results have been achieved. By analyzing the DVM and its support classes we discovered the the encryption mechanism used to protect code. The pass-phrase

used as encryption key followed quickly after that. Another weakness found in the current version is the fact that the tamper-resistance did not yet work. After some fiddling and experimentation we managed to manipulate the decompiled code and recompile it again. This could potentially be used to implement malicious code and launch a reverse engineering attack.

Decompiling the protection technology may sound ironic for a tool that is supposed to protect against reverse engineering, but it is actually a testament to the effectiveness of the protection, because the strength of the protection is not bound to its secrecy. As author of the original programs with full knowledge of the original source code it was still impossible to recover anything that resembled the protected code prior to the virtualization and obfuscation. If there are no weaknesses in the implementation that can be exploited, then the only remaining approach for recovery is a laborious task that entails the reverse engineering of the DVM, and understanding the DVMs program logic before the actual virtualized code can be identified. This was however beyond the scope of our investigation. During the investigation we did however encounter some bugs in the protection technology. It was not always possible to apply protection to our code. Whenever this happened the problem was relayed to the Solidshield developers and they used the feedback to fix the issue. Experimenting with the sorting algorithms and large data sets revealed a bug concerning an overflow of a pointer used internally by the DVM. This caused the pointer to wrap into a negative value when the max value of an integer was exceeded leading to a pointer referring to an out of bounds address. One bug in particular regarding threading was cause for concern. It caused instability and incorrect results. This bug has been solved and it appears to produce stable and functionally equivalent virtualized code since then. The current solidshield version for Java is still under development and there is also some room for improvement judging by our findings, but the bugs discovered and mentioned here were solved quickly in a matter of days.

Chapter 6 presents some important components from the TACTICOS technology stack. Most of these components have been implemented as OSGi bundles in the TACTLESS demo application. OSGi, Logging and Netty have been implemented in bundles. The Netty networking bundle includes some basic Java Swing elements but there is not yet a GUI bundle. The GStreamer framework and JOGL library have therefore not yet been implemented in bundles, because they need a GUI for proper testing. Both are open source projects written in the C programming language. The latter makes Java code virtualization impossible because it is not supported by the Solidshield tool and the former makes it irrelevant as the source code is already part of the public domain. The priority was given to implement Java components from the technology stack that are actual candidates for Java code virtualization. It is still advisable to implement a more elaborate GUI and look into the performance consequences for Gstreamer and JOGL inside a protected application with virtualized code, because the performance overhead of the virtualization might affect graphical performance. The JNA and JNI components have also not been implemented as standalone bundles yet, because they are considered to have a lower priority for similar reasons. Both are part of the public domain. JNI could however be tested implicitly via OpenSplice. Java and Linux are not implemented as bundles, but have been evaluated at a different level. The sorting program and TACTLESS

demo application have been written in Java and they run on a Linux operating system. The OpenSplice bundle has been implemented but does not yet work as intended due to a problem of a technical nature. During the implementation of an OpenSplice bundle a bug has been encountered regarding the use of data-reader-listeners that results in a segmentation fault as shown in Listing 9.1.

```
A fatal error has been detected by the Java Runtime Environment:
#
#  SIGSEGV (0xb) at pc=0x00007f83443dee99, pid=23883, tid
    =140201403762432
#
# JRE version: Java(TM) SE Runtime Environment (8.0_60-b27) (build
    1.8.0_60-b27)
# Java VM: Java HotSpot(TM) 64-Bit Server VM (25.60-b23 mixed mode
    linux-amd64 compressed oops)
# Problematic frame:
# C  [libdcpssaj.so+0x2de99]  saj_dataReaderListenerOnDataAvailable
    +0x39
```

Listing 9.1: OpenSplice workaround

Debugging the problem revealed that the segmentation fault occurred in the *libdcpssaj.so* library. The memory access violation was caused due to the default configuration that sets the service address space to a small value that works fine for C and C++ programs but it is apparently too restrictive for Java programs. This can be remedied by adding the stacksize element in the configuration file with a bigger value. Experimentation showed that a value of 256000 suffices for our tests. Listing 9.2 shows how to address this issue by creating a new XML configuration file or adapting one of the existing configurations in the *OSPl_HOME/etc/config* directory and adding the *StackSize* element.

```
<OpenSplice>
   <Domain>
      <Listeners>
         <StackSize>256000</StackSize>
      </Listeners>
   </Domain>
</OpenSplice>
```

Listing 9.2: OpenSplice workaround

An other OpenSplice restriction posed upon the TACTLESS demo application is related to the specialized DDS/DCPS interfaces that must be generated by the IDL pre-processor. This pre-processor creates TypeSupport, DataReader and DataWriter code from data definitions defined in IDL. These interfaces must be included in the bundle. Communication over OpenSplice has to adhere therefore to the specification defined in IDL. This means that some of the configuration options in the simulator and the implemented protocols are not available when the OpenSplice bundle is used, because varying parameters such as message length and the number of values would violate the specification. Changing some parameters of the protocol would require the generation of new

interfaces by the pre-processor according to a new definition in IDL and rebuild the bundle to include the new interfaces and support classes. This is a cumbersome task and also prevents dynamic changes and manipulation at runtime that would be possible otherwise trough JMX.

Chapter 7 explains why the TACTLESS demo application has been developed. It shows the overall design and motivates the important design decisions. Initially the concept of the OSGi modularity looked very appealing and practical for our development chain. Developing OSGi bundles instead of a typical non-OSGi application confines the developer to best practices for OSGi. The fact that every bundle has its own class loader can be considered a double-edged sword that provides modularity and separation of concerns, but it also has some limits such as patterns that can only be applied inside a bundle and not at an architectural level, because interfaces and classes from other bundles are not visible unless they are exported explicitly. The preferred approach is to use services for interaction but for some components this is tricky, because they are not available as OSGi components and they might use constructs such as static methods or rely on external libraries that are not easily converted to an OSGi implementation. This practical problem also affected our initial design due to the high modularity whereby almost every technology stack component was divided in one or multiple bundles. The initial design for our TACTLESS demo application had a separate networking bundle, Netty bundle, OpenSplice bundle and simulator bundle. Due to some of these practical limitations the simulator and network bundle have been merged. The network bundle has a Netty implementation version and an OpenSplice implementation version. Both versions are quite similar. Only communication is handled differently. The OpenSplice version is however less configurable due to the inflexibility of the fixed datatypes generated by the pre-processor before runtime. An advantage of the modular OSGi approach however is off course the ability to interchange these bundles quite easily and dynamically at runtime when desired.

Chapter 8 presents the findings of evaluating the Solidshield protection with our sorting program and TACTLESS. Benchmarking the sorting algorithms revealed that for the small data sets the runtime is too short for meaningful statements, because the variation in the gathered runtimes is simply to big. Trends are however accurately derived from analyzing the statistically robust collected measurements on the larger data sets. The performance cost of encryption applied to the sorting algorithms is relatively cheap judging by the calculated performance factor, but it is evident that the virtualization protection is very expensive with calculated performance factors over a hundred times worse then the unprotected original implementations. The peak performance factor calculated for BucketSort applied to the 1.000.000 element data set did however not reflect the actual performance trend, because the data set was processed more efficient by the unprotected version then the other data sets. The virtualized BucketSort did not have the same advantage and performed as one might expect based on the results from sorting the other data sets. The optimization specific to that particular data set is every interesting, but the benchmarks and collected metrics offer no clues as to what causes the unprotected BucketSort to benefit from this effect. The unprotected QuickSort also appears to benefit from the same effect to a lesser degree. To rule out a problem with the input

data set the benchmarking has been repeated on ordered data sets. Favorable randomization has been ruled out as a potential cause and it remains unclear what precisely gives the unprotected BucketSort this advantage while the virtualized BucketSort behaves like the projections based on the measurements of the smaller en larger data sets suggest.

During the evaluation of the threaded QuickSortT and QuickSortTT sorting algorithms we encountered a few problems that indicated a problem in the Solidshield DVM. The first problem concerned QuickSortT producing wrong sorting results for the bigger data sets beyond a random point. This was discovered right away thanks to the logging of the md5hash verification applied on the sorted output. Examining the sorted data sets stored on disk revealed that the virtualized QuickSortT did not display functionally equivalent behavior and that obviously posed a serious problem. Investigating the problem we discovered that this non-deterministic behavior was due to a synchronisation issue. We collected the number of recursion calls and counted the number of swap operations performed by the algorithm. These turned out to be inconsistent and unbalanced for the virtualized version. These findings were shared with the Solidshield team and the bug has been fixed. With this fix the output is now correct and the behavior appears to be functionally equivalent. Obfuscation and virtualization transformations producing functional equivalent code do however not necessarily produce code with equal performance, on the contrary, it often leads to performance loss. This was clearly noticeable and reason to experiment with a different threading implementation. The QuickSortTT is a slightly different implementation of our QuickSort algorithm. Both QuickSortT and QuickSortTT perform reasonably similar but once protected with code virtualization there is a significant difference noticeable. The collected metrics also show that QuickSortT and QuickSortTT behave similar, but after virtualization QuickSortT requires more time and memory to run. The cause for this significant difference remains unknown for now. We ruled out overzealous thread instantiating by QuickSortT by limiting the number of QuickSortT threads to the amount of logical cores available on the machine. The QuickSortT has no non-volatile attributes and the parameters are passed as call by value, therefore the local caching inside a thread should not be an issue as the variables remain inside the scope of the method. Read and write actions that require atomicity, such as the swap method, have been synchronized. We performed the tests on several machines with the same results, but they all had the same architecture and that is something that could be looked into, although it is an unlikely candidate for causing these significant differences in results. We observed the effect and ruled out some potential causes but our experiments can not explain the difference. Some coding practices are handled more efficiently by the DVM than others apparently as indicated by our evaluation methodology and the Solidshield developers have better insight and knowledge regarding their product to identify the source of the problem. This is definitely something that should be looked into by the developers. Other important language features such as reflection and JMX have been shown to work after virtualization protection has been applied.

TACTLESS demonstrated that the virtualization applied to OSGi bundles suffers from additional performance overhead. The runtimes for the protected sorting bundles running in an OSGi environment were significantly higher then the protected sorting algorithm from the sorting program. In the literature it

is already suggested that the performance of code virtualization slows down an order of magnitude for every level of interpretation. This is true for the DVM running inside the JVM. In OSGi there are multiple DVMs active, because every protected bundle receives its own DVM. They are however not nested and execute at the same JVM interpretation level. There is however a significant cost for every DVM present. For complex applications with multiple OSGi bundles that require protection this does not scale well.

JMX allowed closer inspection on the matter by collecting metrics from the JVM but there are some limitations to be taken into account when interpreting them. Events that do not last long enough for one sample interval to complete might not be noticed at all. Based on the metrics gathered on the protected BucketSort bundle one might deduce that the DVM does not create additional active threads, because the CPU load tops out at 12,5% and the number of active threads stay the same, but that is not enough. Additional thread statistics were required to ascertain that there is not a thread spawned that synchronizes with the active thread as that would keep the active thread count unchanged. Fortunately it is relatively easy to collect these additional metrics from the JVM via JMX. The live thread count, peak thread count, daemon thread count and total thread count together gives greater certainty in this regard. This still does not rule out the possibility of passing information between threads, but there are no new threads started during the execution of the protected code.

Applying knowledge from the lessons learned during this investigation we managed to achieve a reasonable performance trade-off with TACTLESS bundles under specific circumstances. The Netty networking example discussed shows that selective virtualization of code might produce acceptable results. It must be stressed however that the protected update method still suffers from a significant performance trade-off, but the relatively light method does not severely impact the overall picture in absolute terms. This might be different for more complex methods or processes under strict timing conditions where a hundredth of a second does make a difference, such as in the Netty bundle example when the refresh rate is moved up to 1.000 updates at 1 Hz. Unprotected it manages the refresh rate, but protected the system is unable to complete all updates within the available time window. The Netty networking example shows that there is no universal answer. All depends on the specific situation. What are the requirements, how well does the protected code scale, etc.

There are certainly use cases where the performance trade-off is significant yet acceptable, such as non recurring operations that are executed rarely and not time sensitive. An example of this might be the loading of configuration parameters or calibration algorithms that are only executed during the initialization at start-up of an application and not repeated at runtime.

Chapter 10

Conclusion and Recommendations

This chapter summarizes the thesis, highlights the contribution, mentions the limitations of the current work and proposes opportunities for future research. The chapter is divided into the following sections. Chapter 10.1 contains a summary of the thesis. The main contribution is postulated in Chapter 10.2. Chapter 10.3 mentions limitations of the work as they have been discussed in Chapter 9. Chapter 10.4 gives some recommendations to address these limitations. Chapter 10.5 is used to propose future work. The thesis is concluded in Chapter 10.6 by answering the research questions.

10.1 Summary

This thesis examined the possibility of code virtualization to protect Java bytecode against reverse engineering. The focus lies on the state of the art Solidshield code obfuscation and virtualization tool. The technology is relatively new and therefore the consequences with regard to performance and stability were unknown prior to this work. Sorting algorithms and the TACTLESS demo application have been developed to evaluate the tool.

Chapter 2 contains a literature study on protecting code against reverse engineering. Topics regarding reverse engineering have been investigated and it is evident that Java bytecode is vulnerable to decompilation and unprotected Java programs are therefore very susceptible to reverse engineering. In fact the literature suggest that all programs are vulnerable to reverse engineering to a certain extend. The potential and limitations of current protection strategies such as code obfuscation are explored and code virtualization is briefly introduced as an expensive but effective protection. The problem of evaluating protected Java programs and Java programs in general is addressed. The concept of threat levels are defined to categorize threats and these threat levels are used to rank three protection solutions currently available on the market as an example how current protection techniques fail to prevent reverse engineering attacks.

Chapter 3 introduces the testing and evaluation methodology used to benchmark the performance and validate the behavior of the sorting algorithms and TACTLESS demo application bundles after virtualization. It explains why it is difficult to collect meaningful measurements from Java and how we take this into account by using a robust evaluation approach that allows us to collect and analyze the data in a statistically rigorous manner.

Chapter 4 introduces the sorting algorithms and the sorting program which we used for benchmarking the performance consequences of code virtualization. The sorting algorithms are selected from different complexity classes which is useful to determine the scalability of the virtualization. They are also very suitable to validate the functional equivalency of the protected code because the expected output is known in advance and should be correct.

In Chapter 5 the Solidshield virtualization tool is introduced and dissected. An in-depth analysis of the tool has been performed by reverse engineering the protected sorting program. This has resulted in valuable insights and information not available prior to our investigation. The strengths such as the non-intrusive nature and advanced protection are mentioned, but also the limitations identified such as the performance trade-off and the consequences it has for the development lifecycle in regard to debugging protected code.

Chapter 6 discusses components that have been identified in the TACTICOS technology stack. These components might contain technology that could cause problems in an obfuscated and virtualized program. A selection of these components is implemented in bundles for the TACTLESS demo application which is presented in Chapter 7.

The results presented in Chapter 8 show that there is a significant performance trade-off for code protected by code virtualization, encryption however is relatively cheap. Reflection is an important Java mechanism used extensively in some of the technology stack components. It has been demonstrated to work with code obfuscation and virtualization. Threading however gave some issues. Applying code obfuscation and virtualization to OSGi bundles increases the performance penalty compared to regular non-OSGi code due to the fact that every bundle receives its own DVM whereas the non-OSGi program needs only one DVM if it is distributed in a single JAR file. Although the performance trade-off can be significant for code virtualization protection there are still situations conceivable where the trade-off is acceptable.

10.2 Contribution

This thesis investigated the possibilities of Java code virtualization to protect Java bytecode from reverse engineering. In particular the Solidshield tool has been examined as a possible solution to apply obfuscation and virtualization to protect the TACTICOS CMS developed by Thales against reverse engineering. The statistically rigorous testing approach introduced could be used to evaluate other protection products. Sorting algorithms and the TACTLESS demo application have been developed to test and evaluate the Solidshield protection.

The important components from the TACTICOS technology stack have been identified and implemented in the TACTLESS demo application in a novel way. This demo application can be used to test and evaluate code virtualization or other protection technology without risking to expose intellectual property and trade secrets present in TACTICOS. There were no public performance figures known of code virtualization applied to Java bytecode on the scale of our sorting program and TACTLESS demo application. Our research discovered a few bugs in the tool and led to several recommendations for further improvement.

10.3 Limitations of the Current Work

There are a few limitations in the current work that could not yet be addressed in this thesis. Overall the testing and evaluation approach served the work pretty well, but three observations remain that could not be explained with the collected metrics and benchmark measurements. The performance factor peak or dip, the difference in performance between the protected threading QuickSortT and QuickSortTT implementations, and the deterioration of protected threaded code inside OSGi. The sorting algorithms were very useful to examine the Solidshield virtualization tool, but they are not good representative candidates for code virtualization when applied to a program in realistic working conditions. The TACTLESS demo application has been developed to address this, but some of the technology stack components have not been implemented and it is therefore not yet a representative substitute to a full blown production version of TACTICOS.

10.4 Recommendations

The main improvement that could be realized in our work is extending TACTLESS to include a GUI and the remaining bundles that depend on that GUI. For the Solidshield tool we would like to see the following improvements. The DVM for every JAR file approach should be addressed, because now every OSGi bundle that is protected by Solidshield requires a DVM on its own. That leads to a lot of overhead and does not scale in complex systems such as TACTICOS that contain many of these bundles. A solution that does not require a DVM for every bundle would likely be an improvement in regard to the performance. The anti-tampering protection should be included in the Solidshield version for Java. That would decrease the potential attack surface exploitable for reverse engineering attempts. Also the key-phrase used for encryption should be removed from the DVM. These recommendations have been shared with the Solidshield development team and are being looked into.

10.5 Future Work

Java code virtualization suffers significant performance penalties. This is partly due to the fact that the DVM is an interpreter inside the JVM interpreter. Porting the entire TACTICOS CMS to a compiled language is not realistic. It would therefore be very interesting to investigate the possibility to offload only

sensitive code that needs to be protected in native code and apply code virtualization to that object code instead. This would take the interpretation of the DVM by the JVM out of the equation. We expect this would lead to an optimization that might improve the performance trade-off significantly. Ideally the protected compiled code runs an order of magnitude faster while maintaining the strong protection from code virtualization.

In theory it should be possible to combine Solidshield with other obfuscation product from different vendors. It is however known that the current protection technology used at Thales sometimes gives problems on its own, requiring tweaking and changes to the code before it works properly. Adding Solidshield to the mix would probably not decrease these issues and might be an additional source for conflicts. This should definitely be tested. Then also the question should be raised if the current protection technology should be continued to be used or maybe replaced in its entirety.

One of the design goals kept in mind for TACTLESS was to develop an application that could be handed to external parties without risking exposure of intellectual property and trade secrets from TACTICOS. This also allows external security experts and reverse engineering experts to be involved in attempting to crack the protection and recovering protected code from the TACTLESS demo application. Level 4 experts as defined in the threat pyramid can determine an accurate estimation of the protection strength. Involving a software security firm or reverse engineering company would complete the work.

10.6 Conclusion

Java is a high-level programming language that employs an intermediate bytecode format interpreted by the JVM. Java bytecode is more susceptible to reverse engineering than binary object code from a compiled language and that is a cause for concern. During the literature study it became abundantly clear that all programming languages suffer from this problem to a certain degree. Intellectual property and trade secrets can be recovered relatively easy from unprotected Java bytecode and successful attempts would closely resemble the original source code. Compiled languages require slightly more effort but they can also be disassembled.

There are many approaches that aim to tackle this problem varying from juridical constructions such as patents, copyrights, contracts, licensing, etc. to technical solutions including encryption, obfuscation, remote server side execution, code offloading, etc. None of these approaches offer a perfect solution. There is jurisprudence on how to circumvent legislation designed to prevent reveres engineering. The technical approaches are also limited in their effectiveness.

To protect software from reverse engineering several techniques can be applied. We focused on obfuscation and virtualization as viable techniques for TACTICOS and ruled out server side execution because the TACTICOS CMS has to be operated on naval vessels in a self contained environment and connecting to a remote service does not fit that scenario. Offloading code to a compiled language for better protection is an interesting concept and we looked

briefly into this, but although it takes some additional effort to reverse engineer compiled object code compared to Java bytecode it is not safe either and it has therefore not been pursued further in favor of a promising technique called code virtualization.

Any program protected by a software-based protection solution can be reverse engineered given enough time, resources and determination. Limiting ourselves to the software domain, the best achievable result is to increase the time and effort required for successful reverse engineering to a level that makes it economically unfeasible for potential attackers.

Obfuscation is the concept of transforming a program into a program that has equivalent observable behavior but is more difficult to understand and reverse engineer. Bytecode encryption is the first obfuscation technique we mention in our background study. The concept is flawed as a protection mechanism because the encrypted bytecode has to be decrypted before it can be interpreted by the JVM at some point. Code obfuscation and control flow obfuscation are interesting cost effective techniques to increase the time and effort required for reverse engineering a protected program. There are however also static and dynamic analysis techniques that can help reverse engineer obfuscated programs and combining these techniques can defeat most obfuscation techniques given enough time.

We introduced the threat level pyramid to classify the reverse engineering threats into distinct categories of attackers. For each obfuscation class of the obfuscation techniques discussed in this thesis a protection product has been named as an example and its offered security level rated according to the threat levels defined in the pyramid. Based on the literature and prior research performed at Thales the obfuscation products potency has been assessed. The classification of the mentioned examples and similar products available today are limited to stopping low level threats. This might be perfectly acceptable for programs that run a low risk of being attacked by skilled reverse engineers. The same might be argued for software that becomes obsolete very quickly, such as an app that is quickly replaced by a successor from the newest hype. Then attackers put effort in a task that looses its value rather quickly and when the time required exceeds the economical value the attempt becomes an uneconomical endeavor. TACTICOS however does contain valuable intellectual property and trade secrets and it also has a long lifecycle making it worth while to reverse engineer by potential attackers such as competitors, sub-contractors, clients, nations, etc. As we established, TACTICOS can not operate as a service executed remotely on a server. A naval unit must be able to operate standalone without external dependencies, such as a remote connection to a land based server providing a service. It must therefore be assumed that the production code is distributed to clients and sub-contractors. Competitors and other potential attackers might also lay their hands on production trough clandestine channels. The current protection does not offer enough security to stop the higher level threats from successfully reverse engineering TACTICOS. Code virtualization is however a new solution that might increase the protection level for TACTICOS.

Code virtualization is also a kind of obfuscation and considered to be the most effective and expensive transformation known today. Virtualized code is removed from the original program and translated to a custom bytecode to be interpreted by a custom virtual machine. Decompiling a protected program does not recover the original source code. In order to reverse engineer

protected code the entire bytecode must be analyzed to discover the program logic employed by the custom virtual machine, before the virtualized code can be identified and reconstructed. Code virtualization is successfully applied to protect malware from being discovered and is the toughest modern software protection available according to security researchers. We investigated if code virtualization could also be a viable protection solution on the scale of larger programs such as TACTICOS. By evaluating sorting algorithms in a sorting program and TACTLESS bundles protected by the Solidshield code virtualization tool we determined whether code virtualization is compatible with the TACTICOS technology stack and if the performance is acceptable. Obfuscation transformations can add overhead to the protected program depending on the applied obfuscation techniques. The literature also says virtualization is the toughest and most expensive protection known. Solidshield does virtualization and obfuscation. We examined the tool and investigated the protection applied to our programs and are inclined to agree that there is a significant performance trade-off.

To conclude the thesis we re-translate our findings back into answers on the original research questions posed in Chapter 1.2.

Is Solidshield code virtualization compatible with the TACTICOS technology stack?
Yes, but this is not a clear-cut answer. The effects of virtualization on important language features such as threading, reflection, etc. have been validated during the exploratory experiments on the sorting program and repeated in a different setup with the TACTLESS demo application. Solidshield is a non-intrusive solution and is therefore compatible without requiring modifications to the source code or JVM. There is however an important observation to be made that changing the source code, for methods marked for protection by virtualization, might be advisable to improve their performance.

How should the TACTLESS demo application look like?
The TACTICOS technology stack has been analyzed and the most important components have been identified. The OSGi requirement is satisfied by designing TACTLESS as an OSGi application. Some of the other components are implemented as OSGi bundles.

How does Solidshield code virtualization impact TACTICOS in terms of performance?
There is clearly a performance loss for the protected code. Encryption is a relatively cheap protection but the obfuscation and virtualization have a significant impact. We discovered during our experiments that some operations, such as string concatenation, (nested) loops, function calls, etc. are quite expensive and should be avoided when possible for a better performance trade-off. The choice of threading implementation might make a difference and the established performance trade-off for the sorting program algorithms deteriorates even further for their sorting bundle equivalents inside an OSGi environment.

How does Solidshield code virtualization affect the behavior of TACTICOS in terms of reliability and stability?

Solidshield is still in development and during our investigation we encountered several bugs that caused instability and wrong behavior. These bugs have all been addressed and solved. The performance issue remains however as this is a fundamental consequence of the virtualization.

How effective is code virtualization to prevent reverse engineering of the code?
The studied literature suggests that virtualization is one of the most effective software protections possible. It makes discovering and analyzing malware protected by code virtualization a difficult task.

How secure is the Solidshield protection regarding the prevention of reverse engineering?
The important lessons learned from the literature are that there are no software-based solutions possible to prevent reverse engineering. There are however obfuscation techniques possible that can increase the required effort significantly. Of these techniques code virtualization is considered to be the most effective transformation. Solidshield combines code virtualization with obfuscation and encryption. This is an interesting mix and especially the combination of virtualization and obfuscation makes it a difficult and tedious process to recover the intellectual property that was protected. We estimate that the offered protection can withstand lower level threads and significantly delay reverse engineering attempts up to level 4 threads. For a definitive answer the TACTLESS demo application should be subjected to reverse engineering attacks by level 4 attackers.

> **Research Question:**
> How can code virtualization and code obfuscation provided by Solidshield contribute to the protection against reverse engineering of intellectual property in the Thales Java based technology stack used in their Combat Systems?

Solidshield is an interesting approach to protect Java bytecode with encryption, obfuscation and code virtualization. Encryption is not the strongest protection against reverse engineering, but the performance trade-off is minimal. Code virtualization is the key feature and distinguishes Solidshield from other obfuscation products. Code virtualization is known to be an expensive transformation and it requires a huge effort to break down. The combination of obfuscation and code virtualization inside a DVM, that replaces the protected code with virtual instructions unique for that particular DVM, offers powerful protection that takes a lot of effort to reverse engineer but the performance trade-off is significant. Therefore a comparative assessment must be made to determine which code has to be protected and to what degree, resulting in a hybrid solution where code virtualization combined with obfuscation and encryption is only applied to the most sensitive code containing critical intellectual property, code obfuscation and encryption could be reserved for less critical code and code that does not hold any significance could be limited to encryption. Solidshield is applied to bytecode in a non-intrusive way at the end of the development-chain and does not require changes in the development process. There are however currently some issues in the development version of Solidshield that ought to

be addressed before it can be considered as a viable solution to protect intellectual property in the technology stack used at Thales against reverse engineering.

Bibliography

[1] Jarcrypt, 2015 (accessed December 1th, 2015). `http://sourceforge.net/projects/jarcrypt/`.

[2] Wibu Systems AG. Wibu systems, 2015 (accessed December 1th, 2015). `http://www.wibu.com/digital-content-protection.htm`.

[3] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. *Journal of the ACM (JACM)*, 59(2):6, 2012.

[4] Ira D Baxter and Michael Mehlich. Reverse engineering is reverse forward engineering. In *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*, pages 104–113. IEEE, 1997.

[5] Philippe Beaucamps and Éric Filiol. On the possibility of practically obfuscating programs towards a unified perspective of code protection. *Journal in Computer Virology*, 3(1):3–21, 2007.

[6] Elliot J Chikofsky, James H Cross, et al. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7(1):13–17, 1990.

[7] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[8] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 275–284. ACM, 2011.

[9] Eldad Eilam. *Reversing: secrets of reverse engineering*. John Wiley & Sons, 2011.

[10] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *ACM SIGPLAN Notices*, 42(10):57–76, 2007.

[11] Daniel Jackson and Martin Rinard. Software analysis: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 133–145. ACM, 2000.

[12] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. Java (tm) language specification. *Addisson-Wesley, June*, 2000.

[13] Matthew Karnick, Jeffrey MacBride, Sean McGinnis, Ying Tang, and Ravi Ramachandran. A qualitative analysis of java obfuscation. In *proceedings of 10th IASTED international conference on software engineering and applications, Dallas TX, USA*, 2006.

[14] Johannes Kinder. Towards static analysis of virtualization-obfuscated binaries. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 61–70. IEEE, 2012.

[15] Eric Lafortune. Proguard, 2015 (accessed December 1th, 2015). `http://proguard.sourceforge.net/`.

[16] Van Lindberg. *Intellectual property and open source: a practical guide to protecting code.* " O'Reilly Media, Inc.", 2008.

[17] Matias Madou, Bertrand Anckaert, Bruno De Bus, Koen De Bosschere, Jan Cappaert, and Bart Preneel. On the effectiveness of source code transformations for binary obfuscation. *Sort*, 100:250, 2006.

[18] Nenad Medvidovic, Alexander Egyed, and David S Rosenblum. Round-trip software engineering using uml: From architecture to design and back. In *Proceedings of the Second International Workshop on Object-Oriented Reengineering (WOOR'99), Toulouse, France*, pages 1–8, 1999.

[19] Jan M Memon, Asghar Mughal, Faisal Memon, et al. Preventing reverse engineering threat in java using byte code obfuscation techniques. In *Emerging Technologies, 2006. ICET'06. International Conference on*, pages 689–694. IEEE, 2006.

[20] Jerome Miecznikowski and Laurie Hendren. Decompiling java bytecode: Problems, traps and pitfalls. In *Compiler construction*, pages 111–127. Springer, 2002.

[21] Godfrey Nolan. Introduction. In *Decompiling Java*, pages 1–16. Springer, 2004.

[22] Rolf Rolles. Unpacking virtualization obfuscators. In *3rd USENIX Workshop on Offensive Technologies.(WOOT)*, 2009.

[23] Vladimir Roubtsov. Cracking java byte-code encryption. *URL http://www.javaworld.com/article/2077342/core-java/cracking-java-byte-code-encryption.html. last accessed in October 2015*, 2005.

[24] Mike Strobel. Procyon, java decompiler, 2015 (accessed Ocotber 29th, 2015). `https://bitbucket.org/mstrobel/procyon/wiki/Java%20Decompiler`.

[25] Thales. Tacticos, 2015 (accessed Ocotber 20th, 2015). `https://www.thalesgroup.com/en/worldwide/defence/naval-forces/above-water-warfare/combat-management-systems/tacticos`.

[26] Sharath K Udupa, Saumya K Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. In *Reverse Engineering, 12th Working Conference on*, pages 10–pp. IEEE, 2005.

[27] Egor Ushakov. Fernflower, analytical decompiler for java, 2015 (accessed Ocotber 29th, 2015). `https://github.com/fesh0r/fernflower`.

[28] R Warden. Software reuse and reverse engineering in practice, 1992.

Acronyms

ADS-B Automatic Dependent Surveillance - Broadcast. 52, 53

AIS Automatic Identification System. 52, 53

AOT Ahead of Time Compilation. 84, *Glossary:* AOT

API Application Programming Interface. 45, 51, *Glossary:* API

AWS Above Water Systems. I, *Glossary:* Above Water Systems

CMS Combat Management System. 1, 2, 8, 44, 48, 51

COTS Commercial off-the-shelf. 46, 48, *Glossary:* COTS

CPU Central Processing Unit. 10, 40, 57–59, 62, 66, 67

CSV Comma Separated Values. 53

DCPS Data Centric Publish Subscribe. 47, 49

DDS Data Distribution System. 47–49

DRM Digital Rights Management. 11, 13

DVM Dynamic Virtual Machine. 30–34, 38, 42, 66, 68, 74

GB Gigabyte. 57, 62, 66, 68

GB Gibibyte. 57

GUI Graphical User Interface. 48, 51, 57

HTT Hyper-Threading Technology. 58

IDE Integrated Development Environment. 46, 47, 58

IDL Interface Definition Language. 48, 49

JAR Java Archive. 2, 30, 45, 47, 54, 55, 74

JIT Just In Time Compilation. 22, *Glossary:* JIT

JMX Java Management Extensions. 25, 40, 57, 58

JNA Java Native Access. 13, 50

JNI Java Native Interface. 13, 50, 51

JOGL Java OpenGL. 51

JVM Java Virtual Machine. 2, 3, 10, 13, 17, 22, 25, 26, 36, 40, 44, 45, 50, 53, 57, 58, 73

M2M machine-to-machine. 47

MB Megabyte. 66

NMEA National Marine Electronics Association. 53

OMG Object Management Group. 47

OSGi Open Services Gateway Initiative. 45–47, 52–56, 58, 72–74

SLF4j Simple Logging Facade for Java. 52

TDA Tactical Display Area. 51

UML Unified Modeling Language. 11, 25

URI Uniform Resource Identifier. 49

URL Uniform Resource Locator. 25

WORA Write Once, Run Anywhere. 44

XML Extensible Markup Language. 46, 49

XOR Exclusive-or. 33

Glossary

Above Water Systems A business line of Thales Netherlands that develops and integrates high-tech Command & Control systems for use in the Naval market. I

AOT An approach to translate source code or intermediate code of a high level programming language to machine code whereby the compilation step is performed prior to program execution. 84

API An Application Programming Interface (API) is a particular set of rules and specifications that a software program can follow to access and make use of the services and resources provided by another particular software program that implements that API. 45

bytecode Java bytecode is the instruction set for the Java Virtual Machine.. 2, 3, 7, 9, 10, 12–17, 19, 20, 27, 30, 31, 34, 42, 44, 54

COTS A term used to describe items such as standard products and services available on the commercial marketplace. 46

Java An object-oriented computer programming language that employs an intermediate bytecode format that can be run on the Java Virtual Machine. I, 1–6, 10, 12–22, 25, 27, 30, 31, 34, 38, 42–46, 48, 49, 51, 52, 57

JIT An approach to translate program instructions to machine code whereby the compilation step is performed on runtime while the program is executed. Sometimes referred to as dynamic translation. It is a mix between interpretation and Ahead of Time Compilation (AOT). 22

PrismTech A company that supplies data connectivity solutions, tools and professional services for network communication including OpenSplice which is an implementation of the Object Management Group's Data Distribution Service for Real-Time Systems standerd. 48

TACTICOS An open system architecture, surface ship Command & Control System developed by Thales in use by over twenty leading navies worldwide. I, 1–6, 8, 9, 21, 25, 42–46, 48, 51–54, 57, 58

TACTLESS A demo application used to experiment and test code virtualization with the TACTICOS technology stack without containing any code from TACTICOS. This makes it an ideal research vehicle that is safe to leave the Thales premises, because it does not contain any intelectual property or trade secrets from TACTICOS.. I

Appendix A

TACTLESS Diagrams

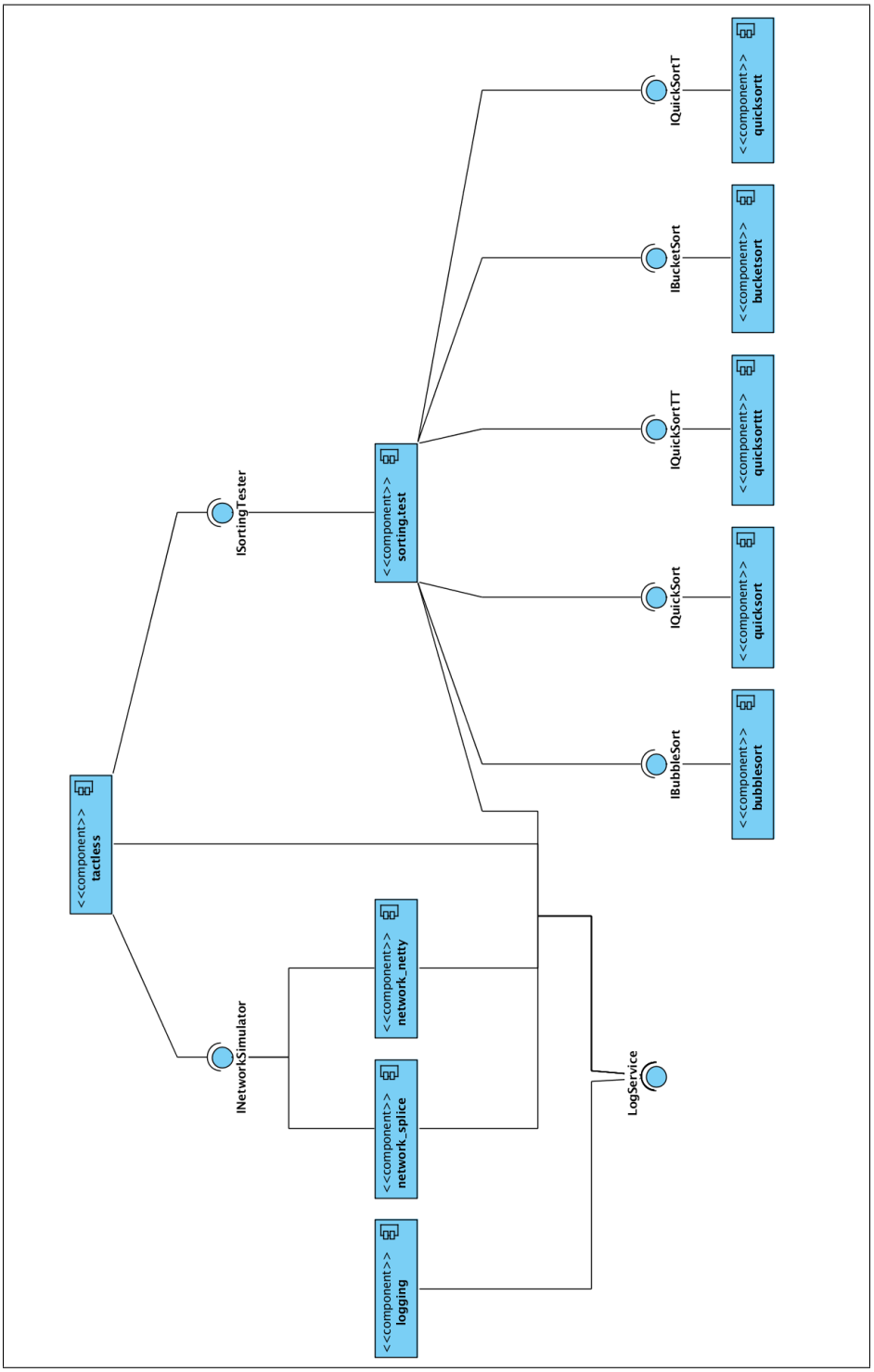

Figure A1: Component diagram TACTLESS.

Appendix B

Runtimes

B.1 Sorting Algorithm Runtimes

	Reference		Encrypted		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$5.83 \cdot 10^{-5}$	$1.1 \cdot 10^{-5}$	$1.34 \cdot 10^{-2}$	$9{\cdot}10^{-3}$	229.9
$1 \cdot 10^2$	$3.72 \cdot 10^{-4}$	$1.6 \cdot 10^{-3}$	$1.99 \cdot 10^{-4}$	$7.8 \cdot 10^{-5}$	$5.4 \cdot 10^{-1}$
$1 \cdot 10^3$	$9.64 \cdot 10^{-4}$	$8.9 \cdot 10^{-6}$	$1.13 \cdot 10^{-3}$	$1.9 \cdot 10^{-4}$	1.2
$1 \cdot 10^4$	0.11	$7{\cdot}10^{-5}$	0.12	$1.6 \cdot 10^{-5}$	1.1
$1 \cdot 10^5$	12.67	$2.1 \cdot 10^{-3}$	12.85	$2.3 \cdot 10^{-4}$	1
$1 \cdot 10^6$	1,252.2	$4.1 \cdot 10^{-1}$	1,267.3	$3.5 \cdot 10^{-4}$	1

Table B1: Encrypted BubbleSort measurements.

	Reference		Virtualized		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$5.83 \cdot 10^{-5}$	$1.1 \cdot 10^{-5}$	$2.09 \cdot 10^{-4}$	$1.1 \cdot 10^{-4}$	3.6
$1 \cdot 10^2$	$3.72 \cdot 10^{-4}$	$1.6 \cdot 10^{-3}$	$7.25 \cdot 10^{-3}$	$2.4 \cdot 10^{-3}$	19.5
$1 \cdot 10^3$	$9.64 \cdot 10^{-4}$	$8.9 \cdot 10^{-6}$	0.71	$6{\cdot}10^{-4}$	732.6
$1 \cdot 10^4$	0.11	$7{\cdot}10^{-5}$	71.1	$7.8 \cdot 10^{-2}$	640.5
$1 \cdot 10^5$	12.67	$2.1 \cdot 10^{-3}$	NaN	NaN	NaN
$1 \cdot 10^6$	1,252.2	$4.1 \cdot 10^{-1}$	NaN	NaN	NaN

Table B2: Virtualized BubbleSort measurements.

	Reference		Encrypted		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$6.9 \cdot 10^{-5}$	$1.8 \cdot 10^{-5}$	$2.4 \cdot 10^{-3}$	$1{\cdot}10^{-2}$	34.9
$1 \cdot 10^2$	$7.25 \cdot 10^{-5}$	$1.5 \cdot 10^{-5}$	$9.86 \cdot 10^{-5}$	$2.9 \cdot 10^{-5}$	1.4
$1 \cdot 10^3$	$7.2 \cdot 10^{-5}$	$1.5 \cdot 10^{-5}$	$1.39 \cdot 10^{-4}$	$3.4 \cdot 10^{-5}$	1.9
$1 \cdot 10^4$	$1.45 \cdot 10^{-4}$	$1.6 \cdot 10^{-5}$	$1.64 \cdot 10^{-4}$	$2{\cdot}10^{-5}$	1.1
$1 \cdot 10^5$	$1{\cdot}10^{-3}$	$3.8 \cdot 10^{-5}$	$1.02 \cdot 10^{-3}$	$6.5 \cdot 10^{-5}$	1
$1 \cdot 10^6$	$2.97 \cdot 10^{-3}$	$1.4 \cdot 10^{-5}$	$2.96 \cdot 10^{-3}$	$3.4 \cdot 10^{-5}$	$1{\cdot}10^{0}$
$1 \cdot 10^7$	0.16	$1.3 \cdot 10^{-4}$	0.16	$4.6 \cdot 10^{-4}$	$9.9 \cdot 10^{-1}$
$1 \cdot 10^8$	1.99	$2.1 \cdot 10^{-3}$	1.98	$2.6 \cdot 10^{-3}$	$9.9 \cdot 10^{-1}$
$1 \cdot 10^9$	15.79	$9.7 \cdot 10^{-2}$	16.46	5.8	1

Table B3: Encrypted BucketSort measurements.

	Reference		Virtualized		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$6.9 \cdot 10^{-5}$	$1.8 \cdot 10^{-5}$	$1.9 \cdot 10^{-4}$	$1{\cdot}10^{-4}$	2.8
$1 \cdot 10^2$	$7.25 \cdot 10^{-5}$	$1.5 \cdot 10^{-5}$	$4.37 \cdot 10^{-4}$	$6.2 \cdot 10^{-4}$	6
$1 \cdot 10^3$	$7.2 \cdot 10^{-5}$	$1.5 \cdot 10^{-5}$	$2.13 \cdot 10^{-3}$	$7.9 \cdot 10^{-4}$	29.5
$1 \cdot 10^4$	$1.45 \cdot 10^{-4}$	$1.6 \cdot 10^{-5}$	$1.73 \cdot 10^{-2}$	$2.7 \cdot 10^{-5}$	119.3
$1 \cdot 10^5$	$1{\cdot}10^{-3}$	$3.8 \cdot 10^{-5}$	0.17	$6.2 \cdot 10^{-4}$	172.1
$1 \cdot 10^6$	$2.97 \cdot 10^{-3}$	$1.4 \cdot 10^{-5}$	1.72	$5{\cdot}10^{-4}$	578.9
$1 \cdot 10^7$	0.16	$1.3 \cdot 10^{-4}$	17.94	$6.3 \cdot 10^{-3}$	110.9
$1 \cdot 10^8$	1.99	$2.1 \cdot 10^{-3}$	180.29	$2.1 \cdot 10^{-1}$	90.4
$1 \cdot 10^9$	15.79	$9.7 \cdot 10^{-2}$	1,857.53	$6.6 \cdot 10^{-1}$	117.7

Table B4: Virtualized BucketSort measurements.

	Reference		Encrypted		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$7.74 \cdot 10^{-5}$	$4.2 \cdot 10^{-5}$	$2.51 \cdot 10^{-3}$	$9{\cdot}10^{-3}$	32.4
$1 \cdot 10^2$	$7.69 \cdot 10^{-5}$	$2.1 \cdot 10^{-5}$	$1.64 \cdot 10^{-4}$	$7.8 \cdot 10^{-5}$	2.1
$1 \cdot 10^3$	$2.35 \cdot 10^{-4}$	$5.8 \cdot 10^{-4}$	$2.75 \cdot 10^{-4}$	$1.9 \cdot 10^{-4}$	1.2
$1 \cdot 10^4$	$8.31 \cdot 10^{-4}$	$3.5 \cdot 10^{-5}$	$1.39 \cdot 10^{-3}$	$1.6 \cdot 10^{-5}$	1.7
$1 \cdot 10^5$	$8.88 \cdot 10^{-3}$	$3{\cdot}10^{-5}$	$1.51 \cdot 10^{-2}$	$2.3 \cdot 10^{-4}$	1.7
$1 \cdot 10^6$	$7.51 \cdot 10^{-2}$	$7.5 \cdot 10^{-5}$	0.12	$3.5 \cdot 10^{-4}$	1.7
$1 \cdot 10^7$	1.14	$1.3 \cdot 10^{-2}$	1.71	$4.8 \cdot 10^{-4}$	1.5
$1 \cdot 10^8$	12.98	$2.4 \cdot 10^{-3}$	18.75	$2{\cdot}10^{-2}$	1.4
$1 \cdot 10^9$	131.22	$1.1 \cdot 10^{-1}$	187.52	10.4	1.4

Table B5: Encrypted QuickSort measurements.

	Reference		Virtualized		
elements	(s)	σ	(s)	σ	$factor$
$1 \cdot 10^1$	$7.74 \cdot 10^{-5}$	$4.2 \cdot 10^{-5}$	$5.48 \cdot 10^{-4}$	$1.3 \cdot 10^{-4}$	7.1
$1 \cdot 10^2$	$7.69 \cdot 10^{-5}$	$2.1 \cdot 10^{-5}$	$9.19 \cdot 10^{-4}$	$6.6 \cdot 10^{-4}$	11.9
$1 \cdot 10^3$	$2.35 \cdot 10^{-4}$	$5.8 \cdot 10^{-4}$	$1.04 \cdot 10^{-2}$	$2.2 \cdot 10^{-2}$	44.1
$1 \cdot 10^4$	$8.31 \cdot 10^{-4}$	$3.5 \cdot 10^{-5}$	$7.39 \cdot 10^{-2}$	$5.2 \cdot 10^{-5}$	88.9
$1 \cdot 10^5$	$8.88 \cdot 10^{-3}$	$3 \cdot 10^{-5}$	0.9	$3.4 \cdot 10^{-4}$	101.6
$1 \cdot 10^6$	$7.51 \cdot 10^{-2}$	$7.5 \cdot 10^{-5}$	9.71	$4.9 \cdot 10^{-3}$	129.4
$1 \cdot 10^7$	1.14	$1.3 \cdot 10^{-2}$	117.39	$5.7 \cdot 10^{-2}$	103
$1 \cdot 10^8$	12.98	$2.4 \cdot 10^{-3}$	1,323.69	$3.4 \cdot 10^{-4}$	102
$1 \cdot 10^9$	131.22	$1.1 \cdot 10^{-1}$	14,246.17	1.5	108.6

Table B6: Virtualized QuickSort measurements.

B.2 Threaded Runtimes

	Reference		Virtualized		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$1.45 \cdot 10^{-4}$	$3.2 \cdot 10^{-5}$	$8.28 \cdot 10^{-4}$	$7.3 \cdot 10^{-5}$	5.7
$1 \cdot 10^2$	$2.12 \cdot 10^{-4}$	$2.8 \cdot 10^{-5}$	$4.15 \cdot 10^{-3}$	$1.3 \cdot 10^{-4}$	19.6
$1 \cdot 10^3$	$1.68 \cdot 10^{-4}$	$1.5 \cdot 10^{-5}$	$3.68 \cdot 10^{-2}$	$1.3 \cdot 10^{-3}$	219.4
$1 \cdot 10^4$	$3.6 \cdot 10^{-4}$	$2 \cdot 10^{-5}$	0.39	$1.3 \cdot 10^{-2}$	1,091.5
$1 \cdot 10^5$	$2.81 \cdot 10^{-3}$	$1.8 \cdot 10^{-4}$	5.48	$9.3 \cdot 10^{-2}$	1,947.4
$1 \cdot 10^6$	$3.59 \cdot 10^{-2}$	$2.3 \cdot 10^{-5}$	144.87	1.8	4,037.8
$1 \cdot 10^7$	0.27	$1 \cdot 10^{-2}$	576.3	7.8	2,102.2
$1 \cdot 10^8$	3.06	$9 \cdot 10^{-2}$	NaN	NaN	NaN

Table B7: Virtualized QuickSortT measurements.

	Reference		Virtualized		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$3.33 \cdot 10^{-4}$	$1 \cdot 10^{-4}$	$2.85 \cdot 10^{-3}$	$1.1 \cdot 10^{-3}$	8.6
$1 \cdot 10^2$	$2.74 \cdot 10^{-4}$	$5.1 \cdot 10^{-5}$	$7.84 \cdot 10^{-3}$	$1.2 \cdot 10^{-3}$	28.6
$1 \cdot 10^3$	$2.71 \cdot 10^{-4}$	$5.5 \cdot 10^{-5}$	$1.49 \cdot 10^{-2}$	$1.5 \cdot 10^{-3}$	55.1
$1 \cdot 10^4$	$4.62 \cdot 10^{-4}$	$3.7 \cdot 10^{-5}$	$6.33 \cdot 10^{-2}$	$2.1 \cdot 10^{-3}$	137.1
$1 \cdot 10^5$	$3.13 \cdot 10^{-3}$	$2.9 \cdot 10^{-4}$	0.55	$2.4 \cdot 10^{-2}$	175.9
$1 \cdot 10^6$	$2.8 \cdot 10^{-2}$	$1.3 \cdot 10^{-3}$	7.14	$1.5 \cdot 10^{-1}$	255.4
$1 \cdot 10^7$	0.22	$1 \cdot 10^{-2}$	53.68	1.3	238.9
$1 \cdot 10^8$	2.25	$8 \cdot 10^{-2}$	512.51	7.7	228

Table B8: Virtualized QuickSortTT measurements.

Appendix C

Performance

C.1 Sorting Algorithm Performance Factor

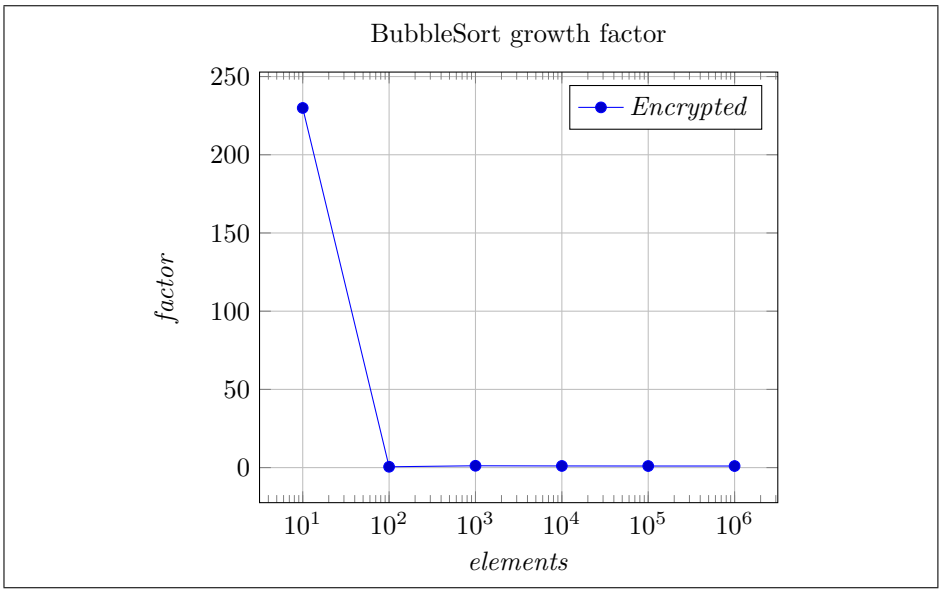

Figure C1: Growth factor encrypted BubbleSort.

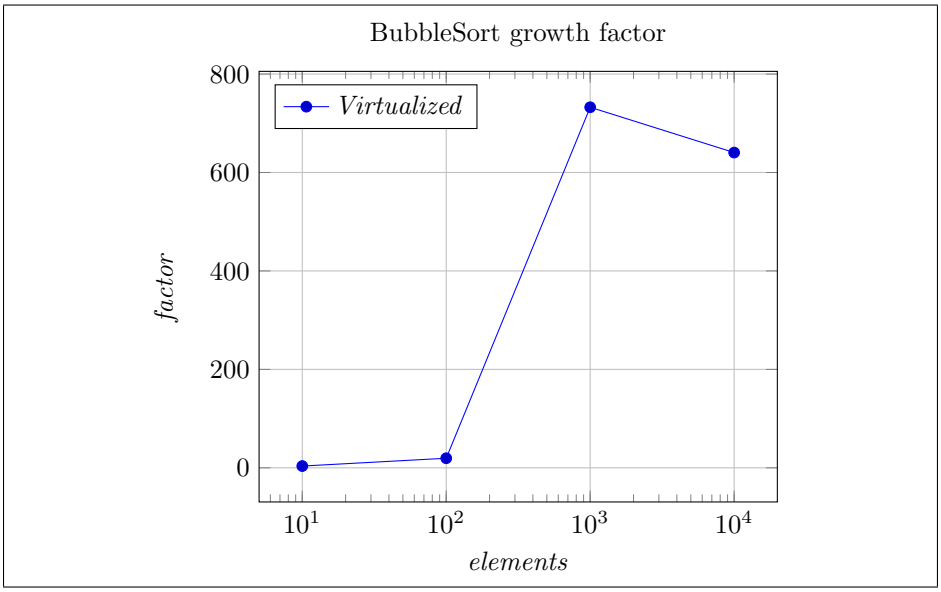

Figure C2: Growth factor virtualized BubbleSort.

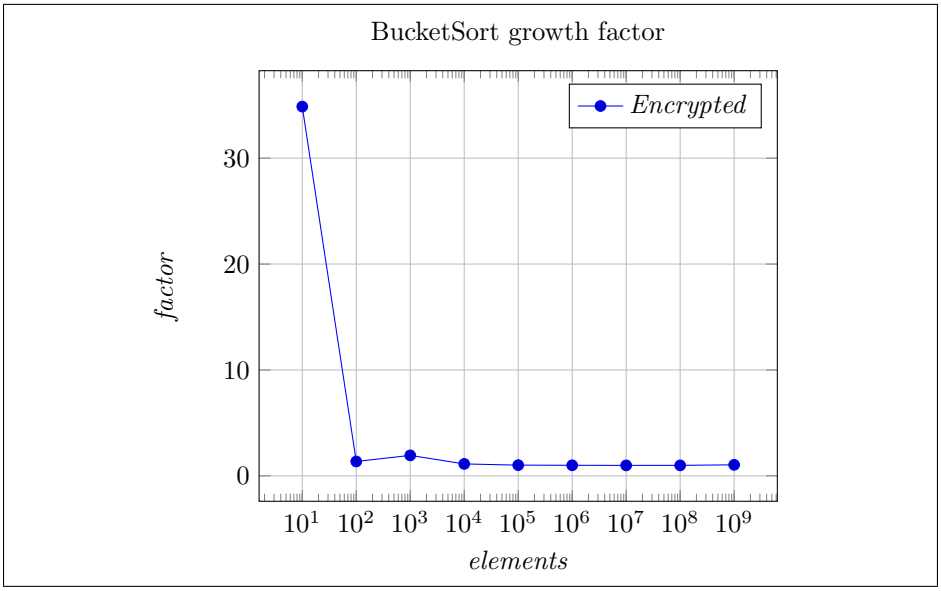

Figure C3: Growth factor encrypted BucketSort.

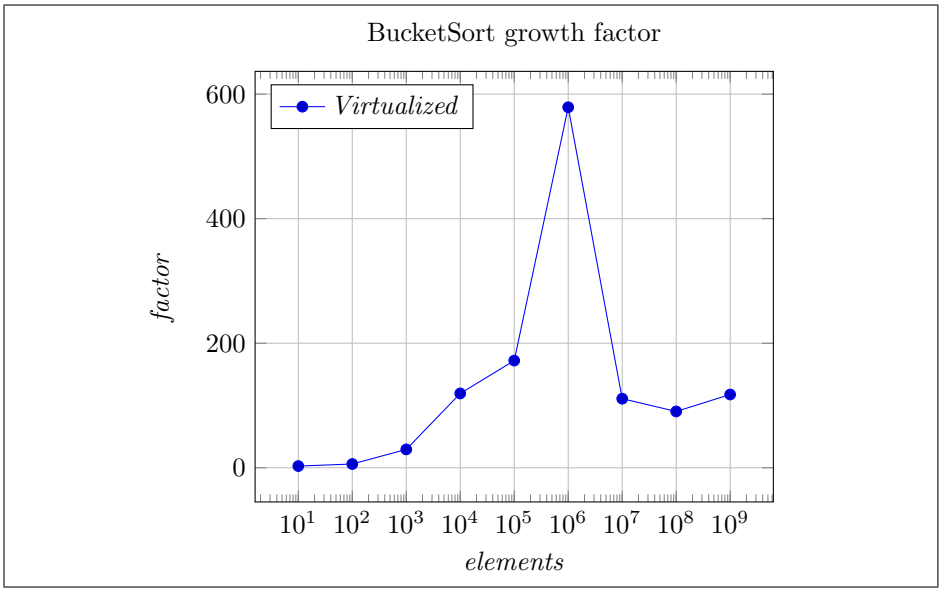

Figure C4: Growth factor virtualized BucketSort.

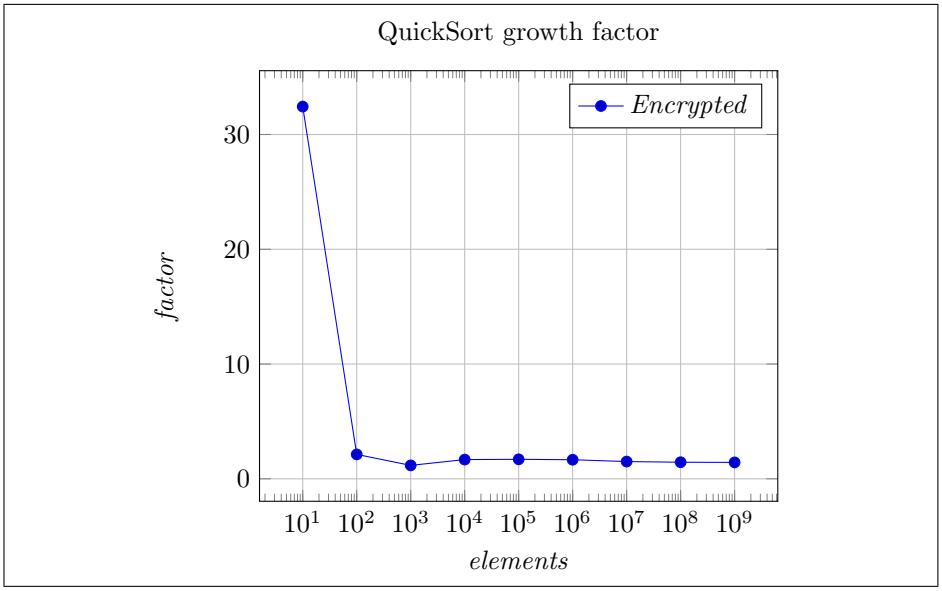

Figure C5: Growth factor encrypted QuickSort.

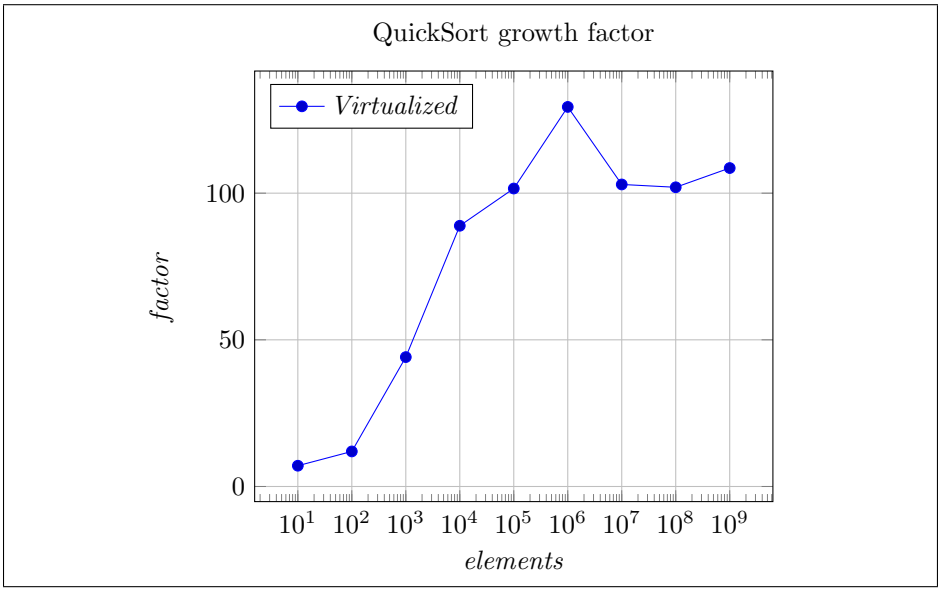

Figure C6: Growth factor virtualized QuickSort.

C.2 Threaded Performance Factor

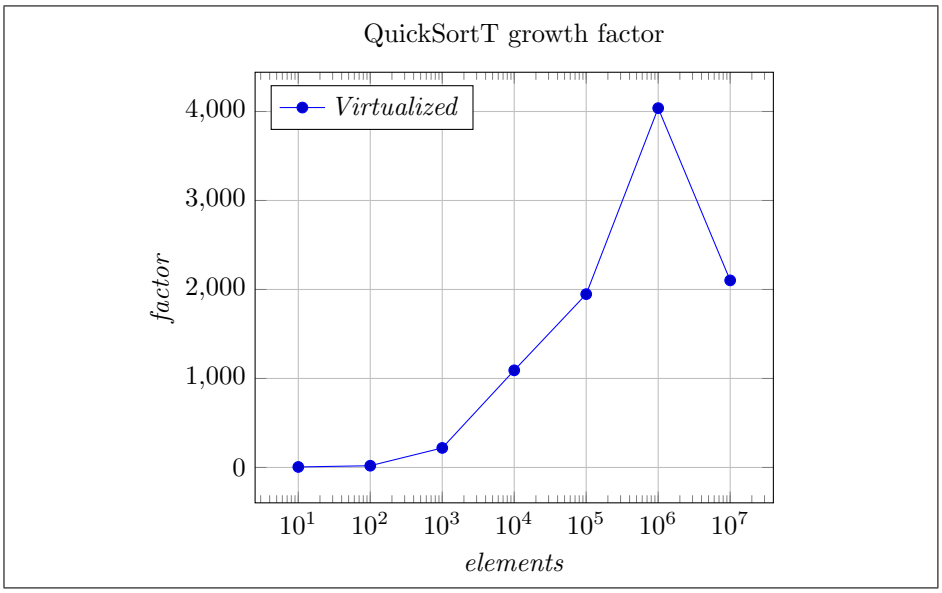

Figure C7: Growth factor virtualized QuickSortT.

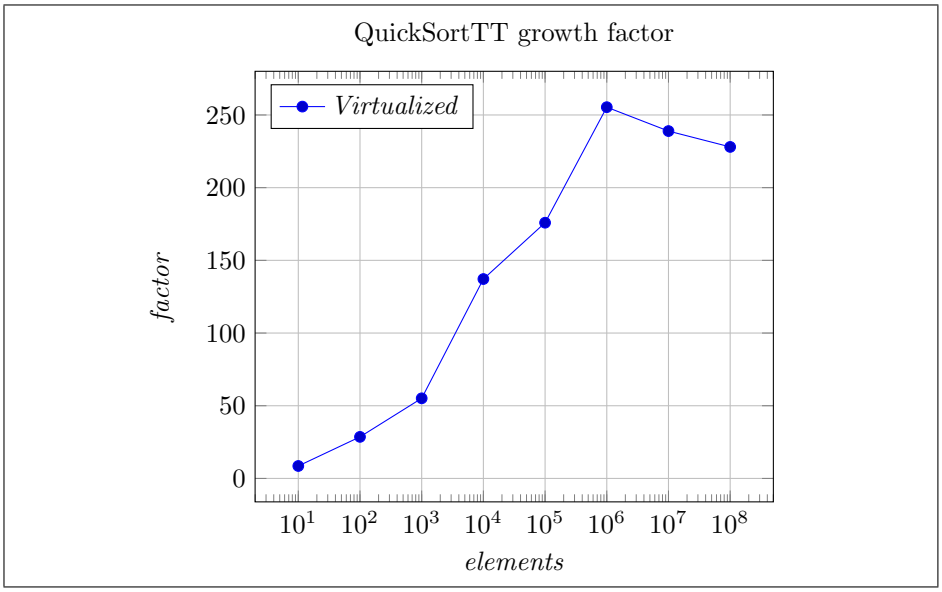

Figure C8: Growth factor virtualized QuickSortTT.

Appendix D

Measurements

D.1 Sorting Algorithm Measurements

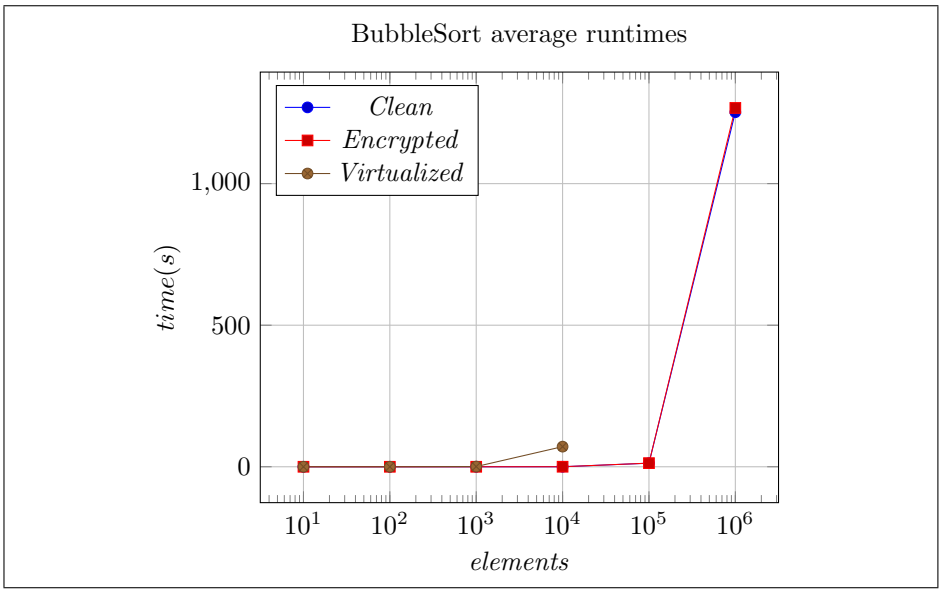

Figure D1: Average BubbleSort measurements in seconds

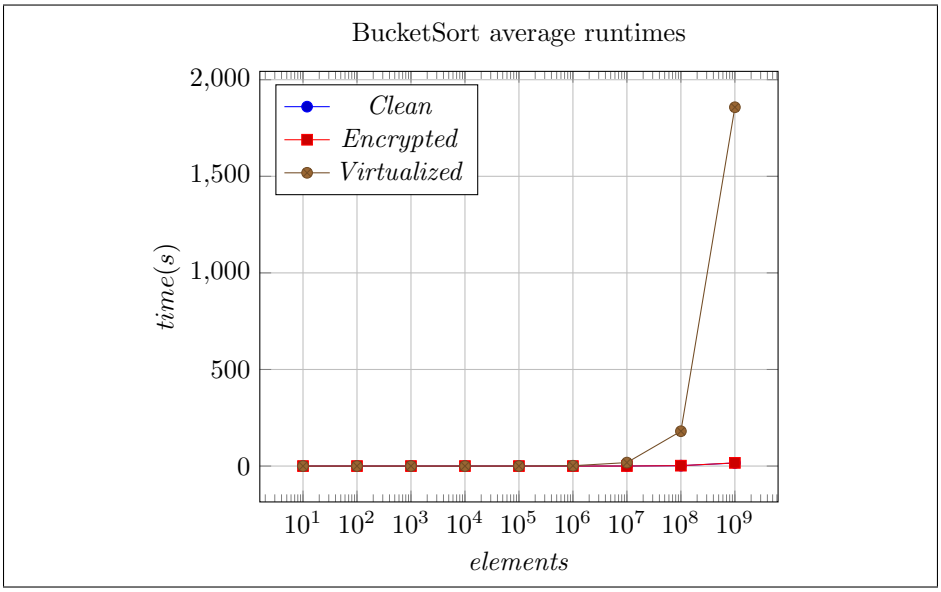

Figure D2: Average BucketSort measurements in seconds

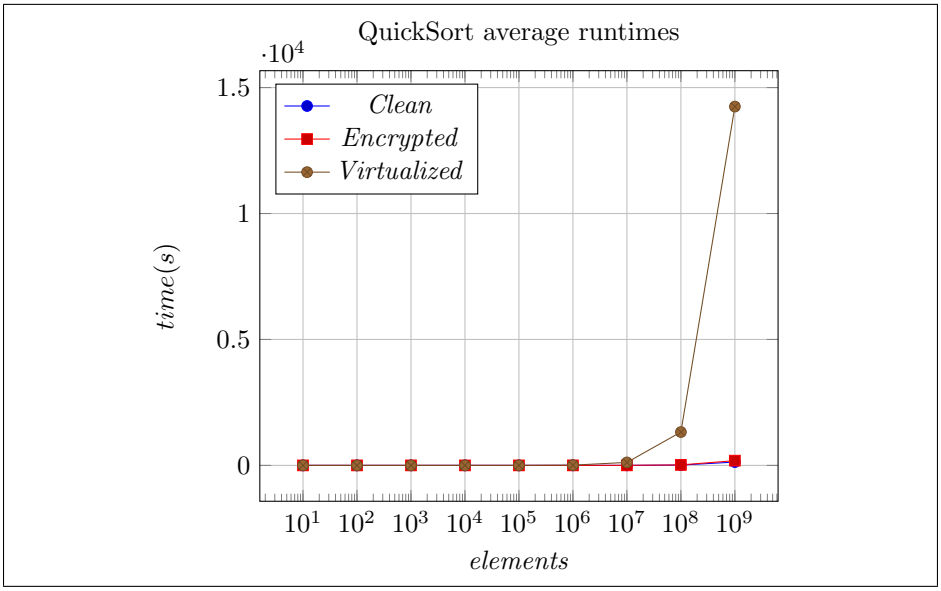

Figure D3: Average QuickSort measurements in seconds

D.2 Threaded Measurements

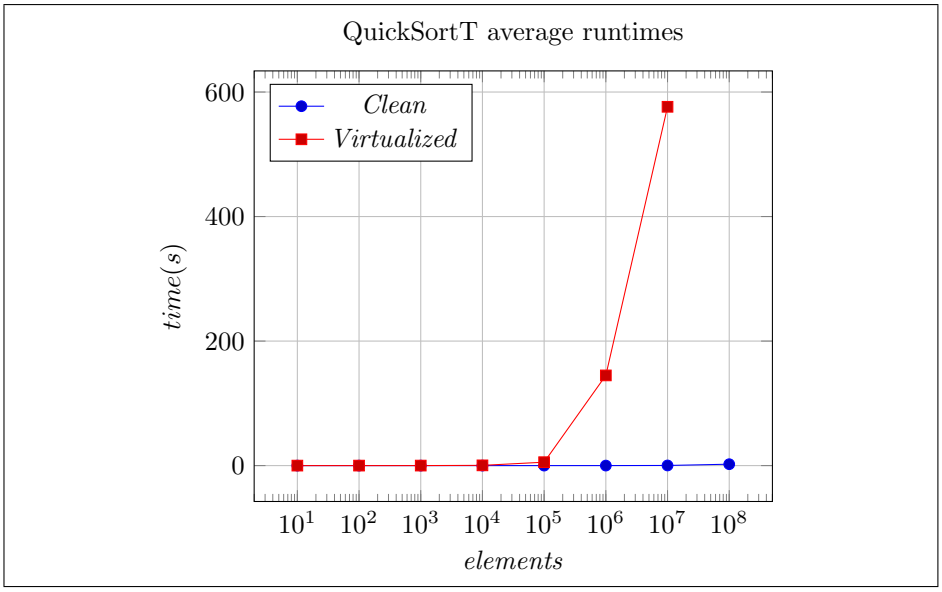

Figure D4: Average QuickSortT measurements in seconds

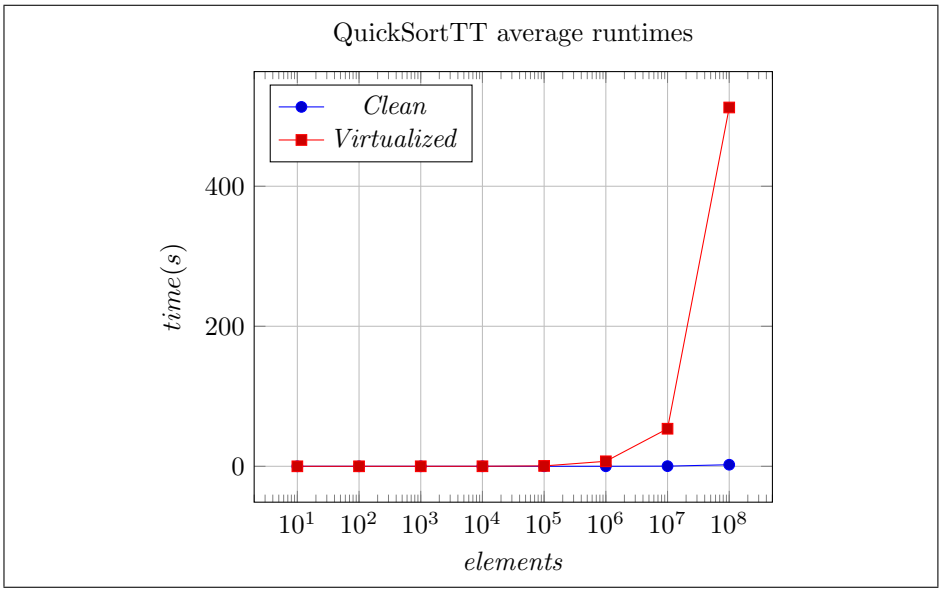

Figure D5: Average QuickSortTT measurements in seconds

Appendix E

Sorting Algorithm Scatter Plots

E.1 Original Reference Measurements

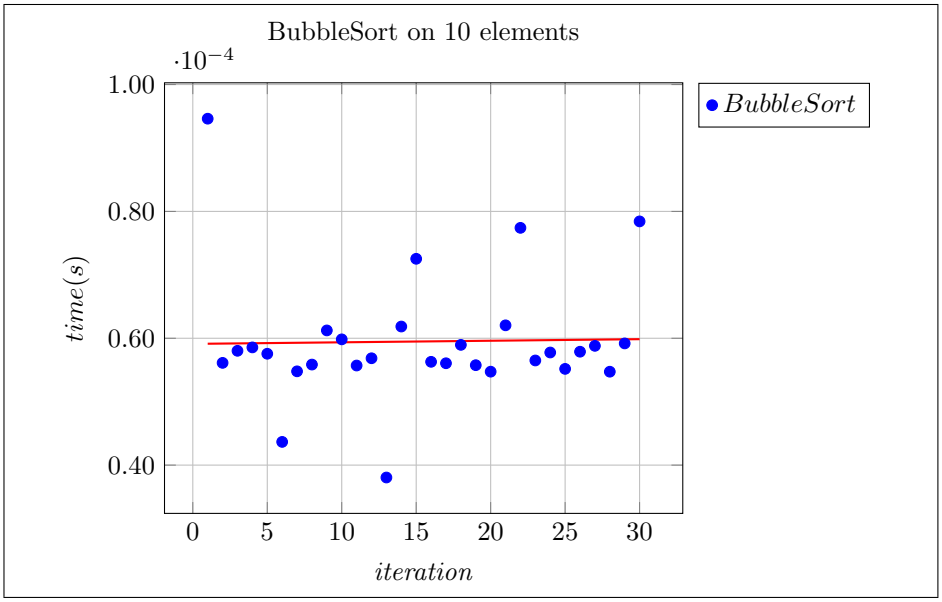

Figure E1: BubbleSort scatter plot on 10 elements.

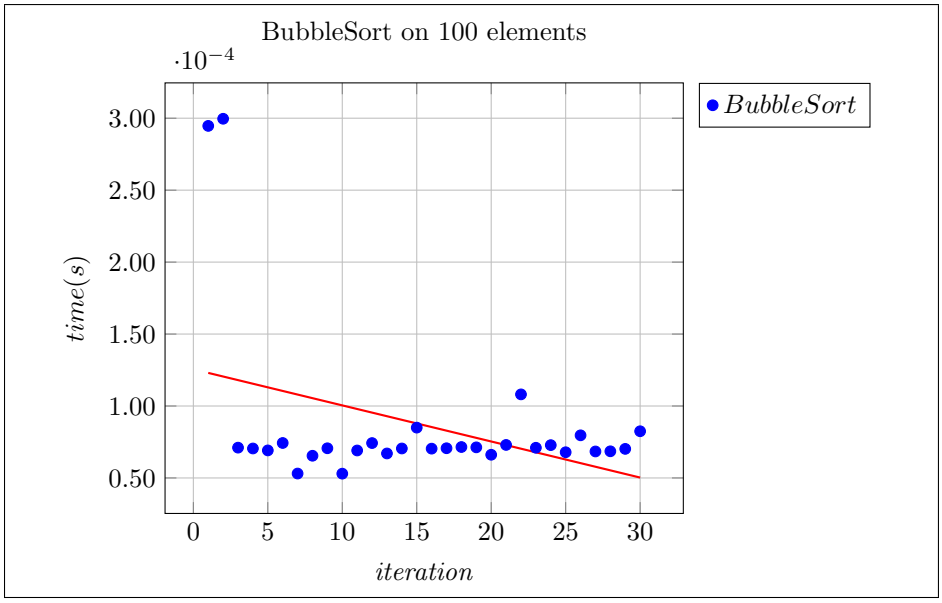

Figure E2: BubbleSort scatter plot on 100 elements with regression line.

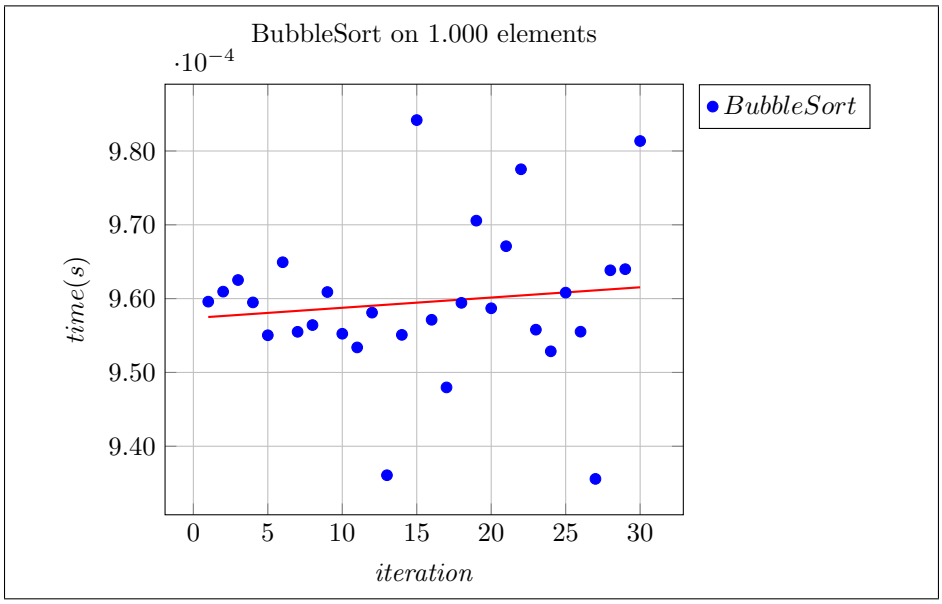

Figure E3: BubbleSort scatter plot on 1.000 elements.

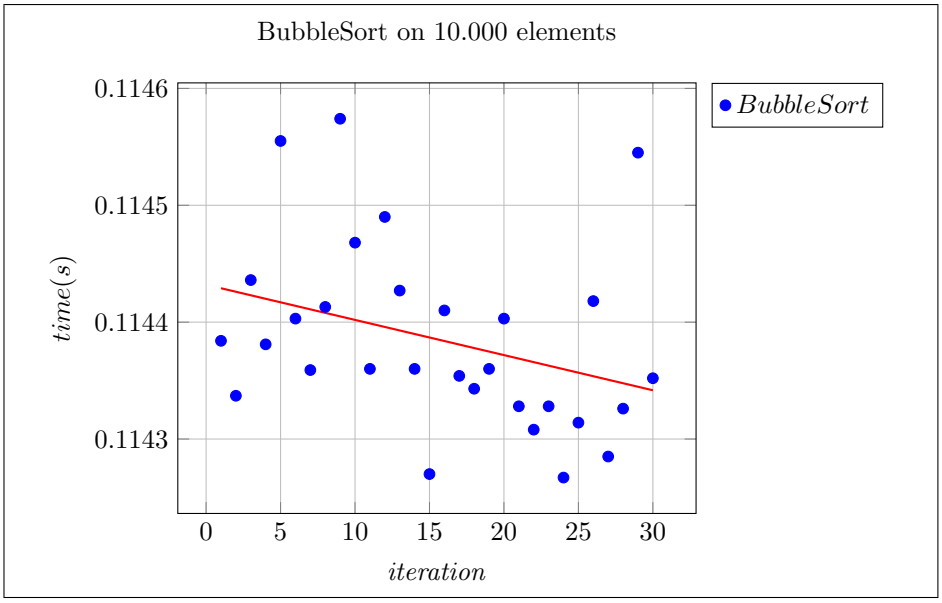

Figure E4: BubbleSort scatter plot on 10.000 elements.

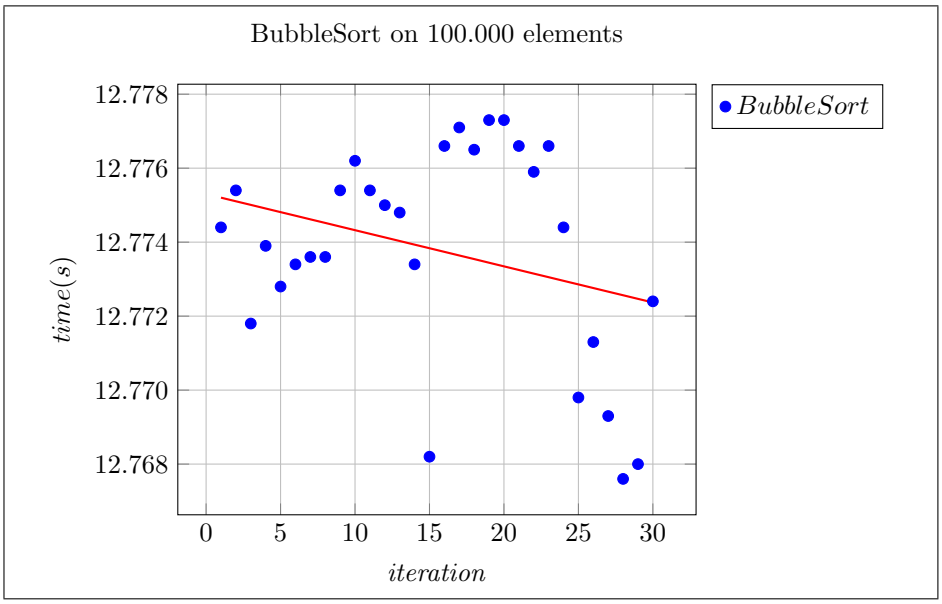

Figure E5: BubbleSort scatter plot on 100.000 elements.

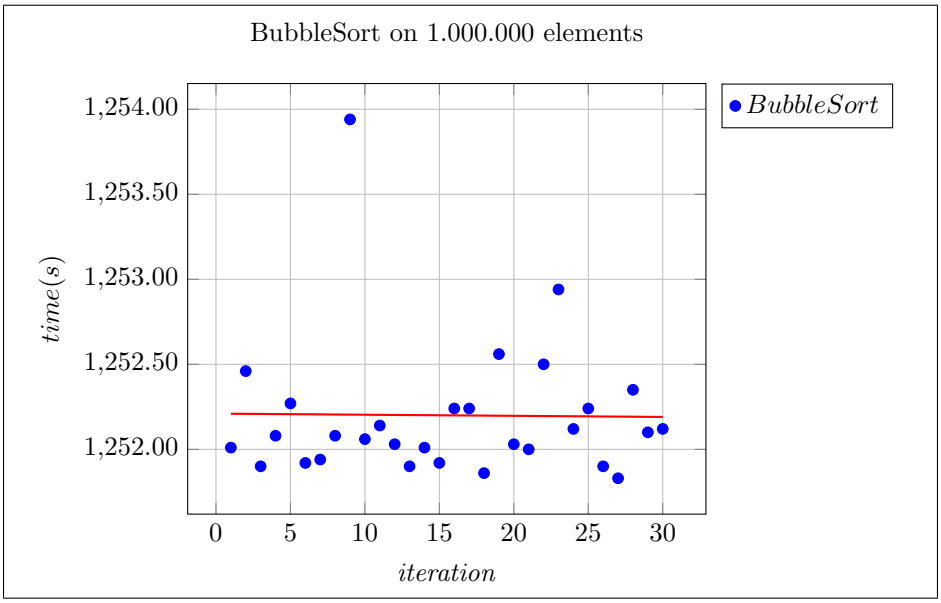

Figure E6: BubbleSort scatter plot on 1.000.000 elements.

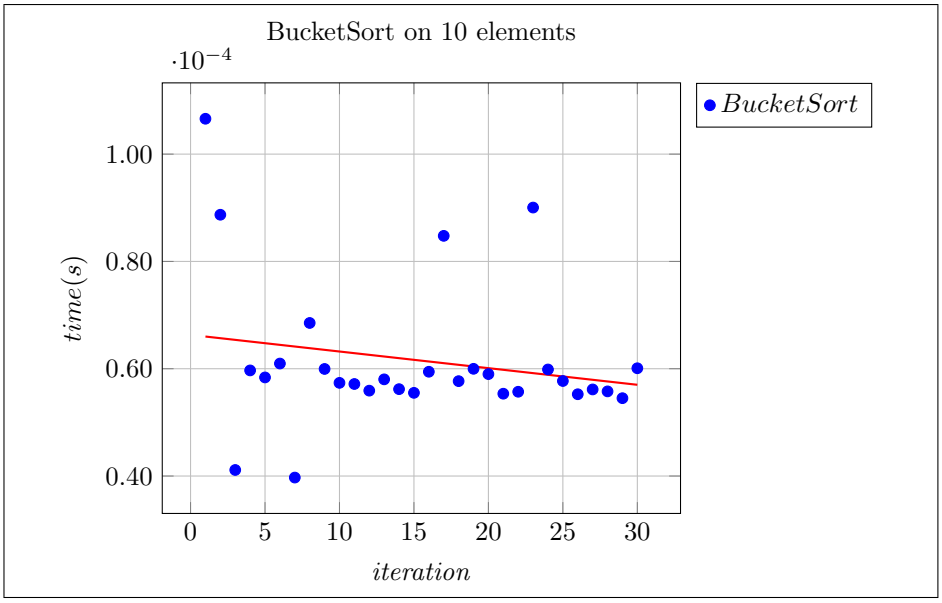

Figure E7: BucketSort scatter plot on 10 elements.

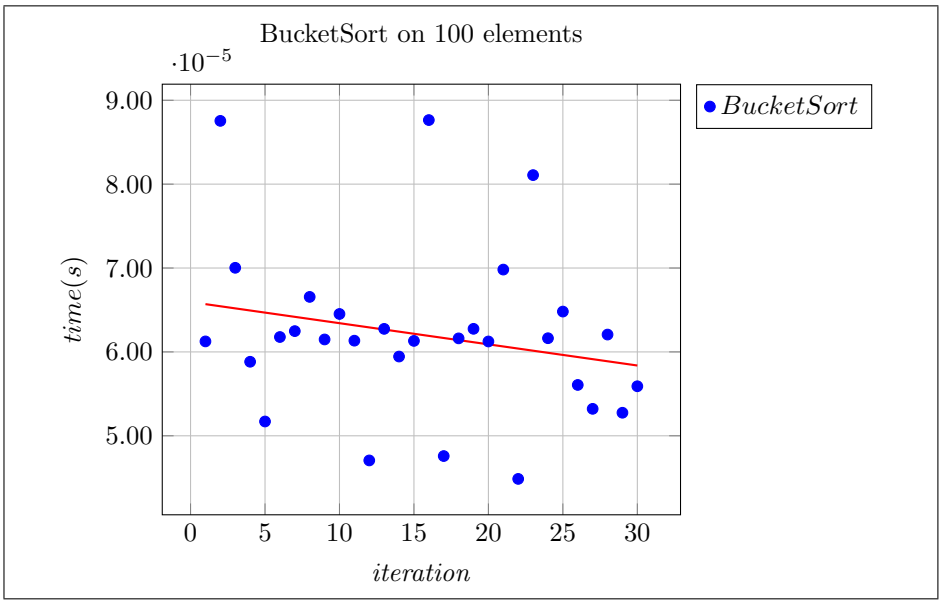

Figure E8: BucketSort scatter plot on 100 elements.

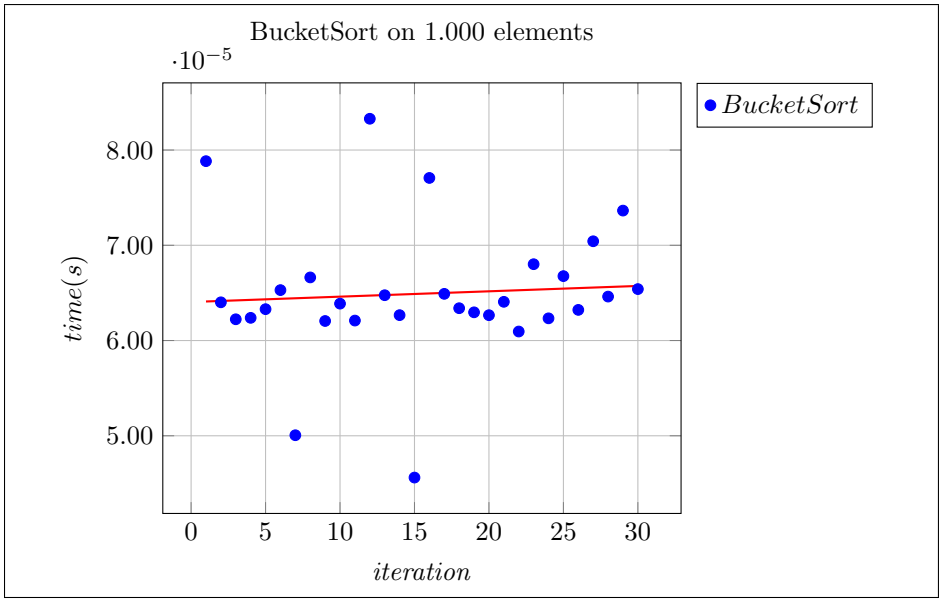

Figure E9: BucketSort scatter plot on 1.000 elements.

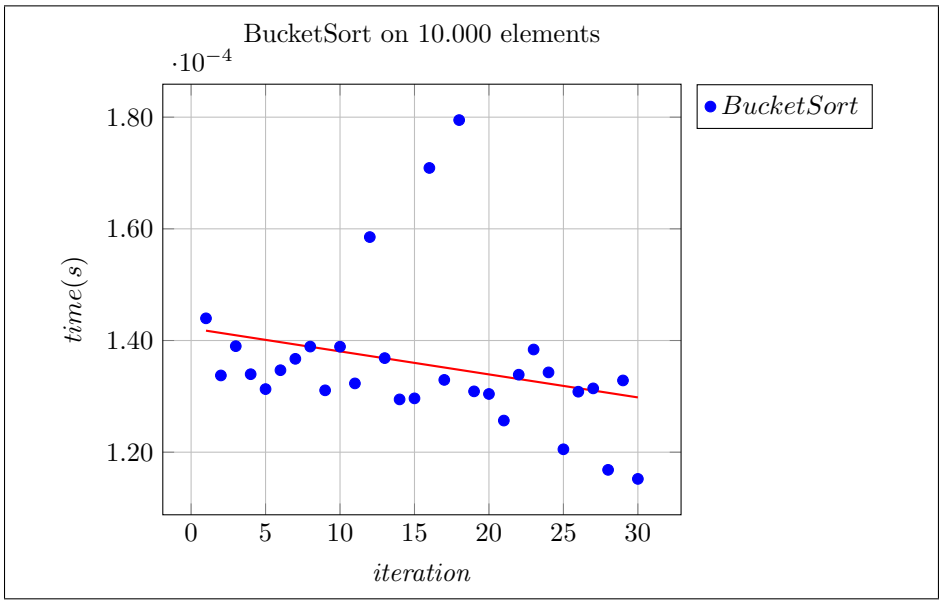

Figure E10: BucketSort scatter plot on 10.000 elements.

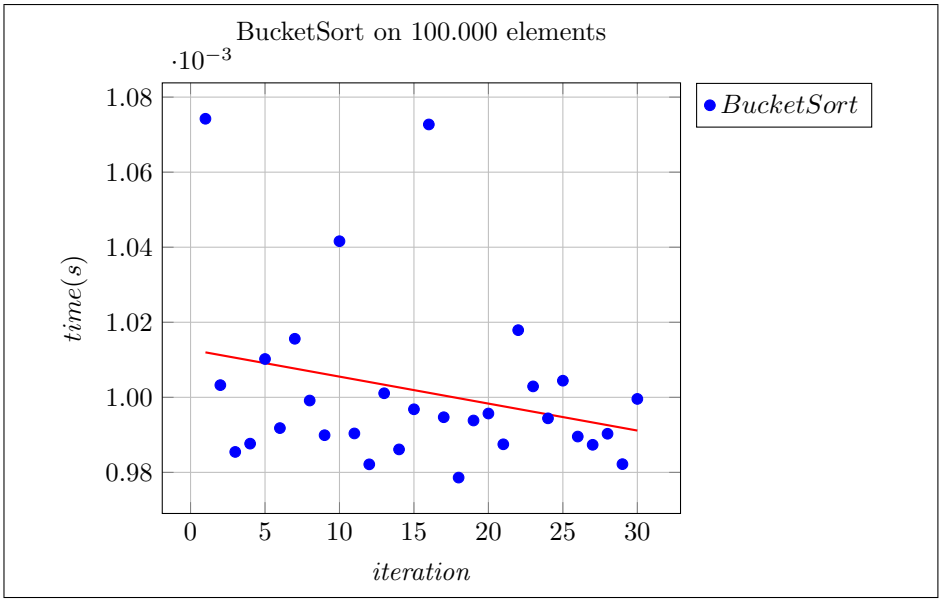

Figure E11: BucketSort scatter plot on 100.000 elements.

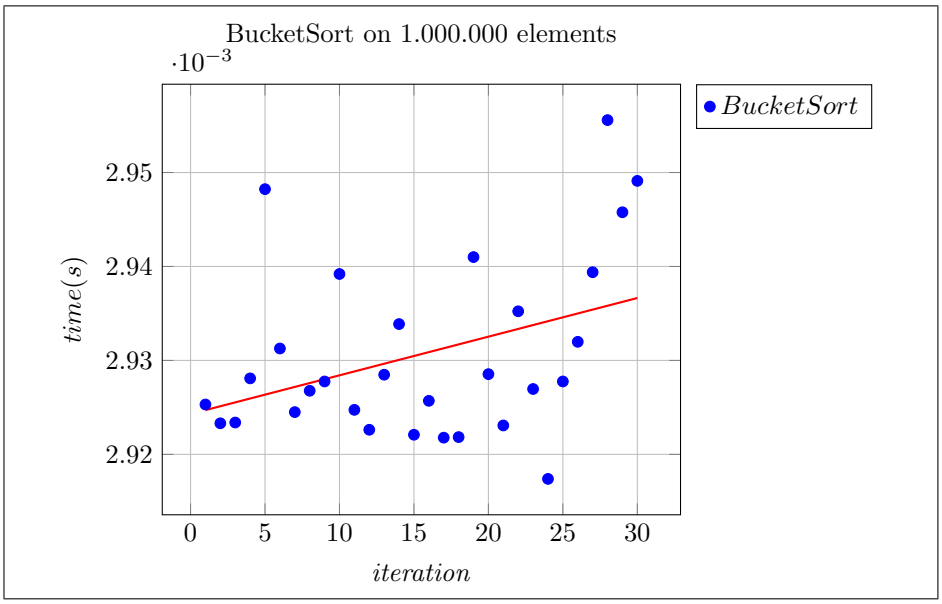

Figure E12: BucketSort scatter plot on 1.000.000 elements.

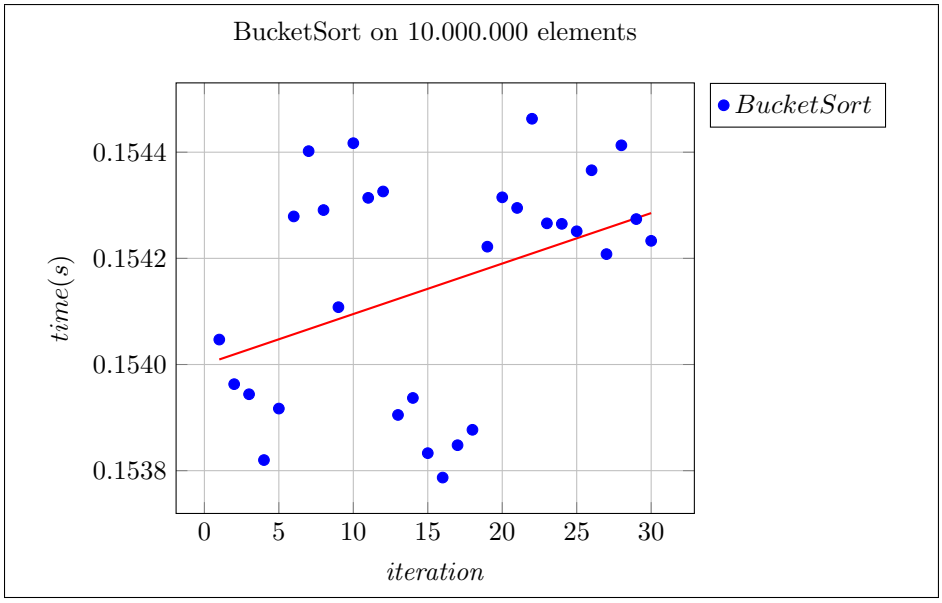

Figure E13: BucketSort scatter plot on 10.000.000 elements.

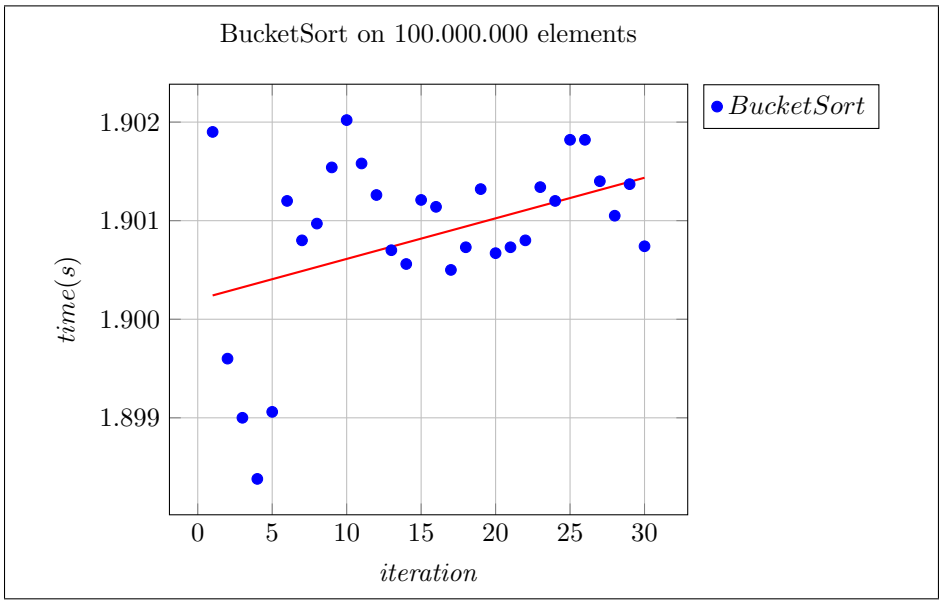

Figure E14: BucketSort scatter plot on 100.000.000 elements.

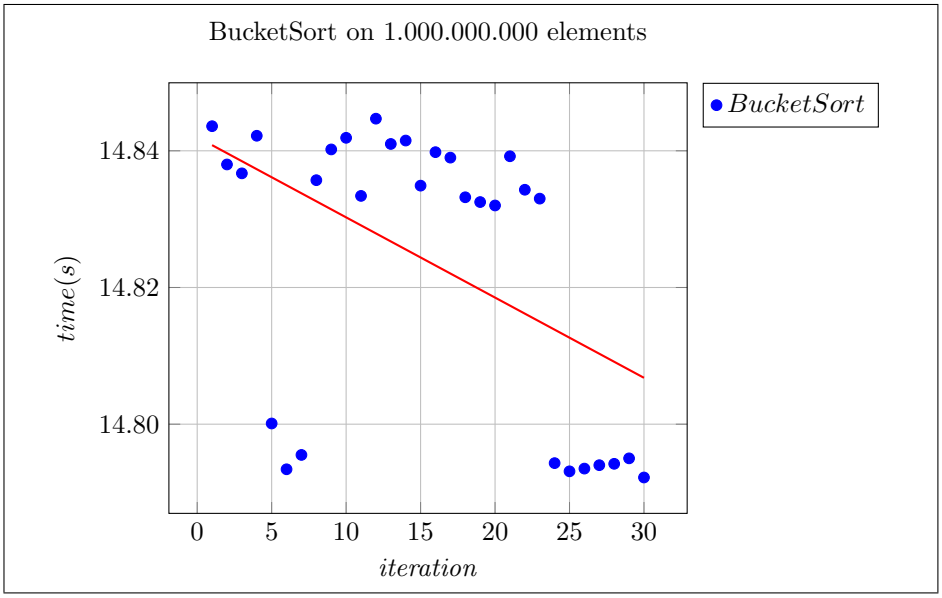

Figure E15: BucketSort scatter plot on 1.000.000.000 elements.

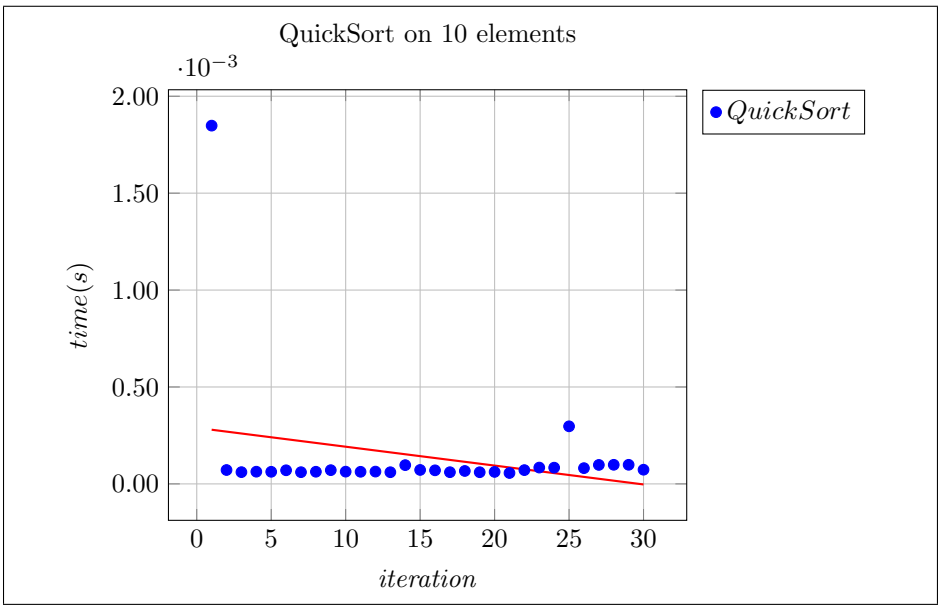

Figure E16: QuickSort scatter plot on 10 elements.

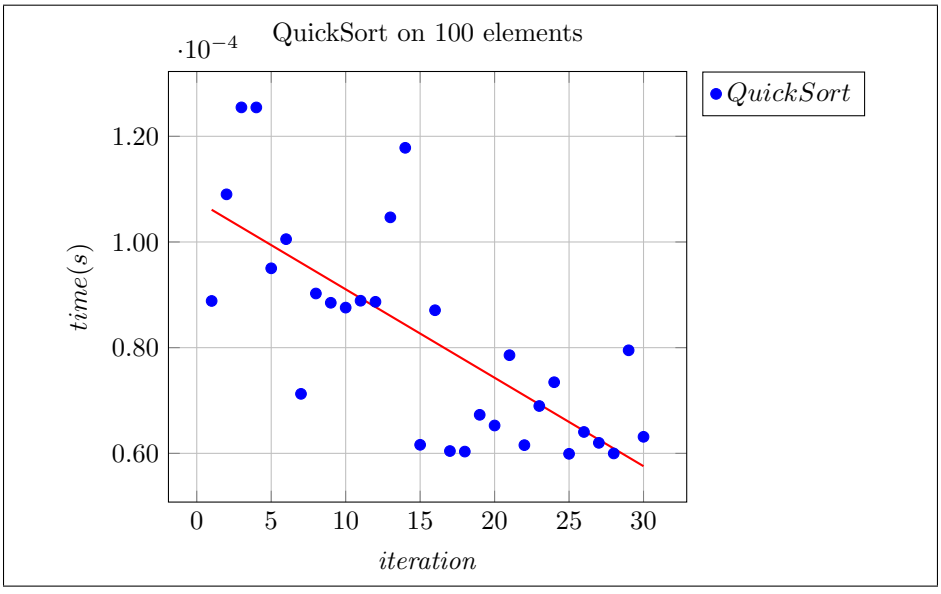

Figure E17: QuickSort scatter plot on 100 elements.

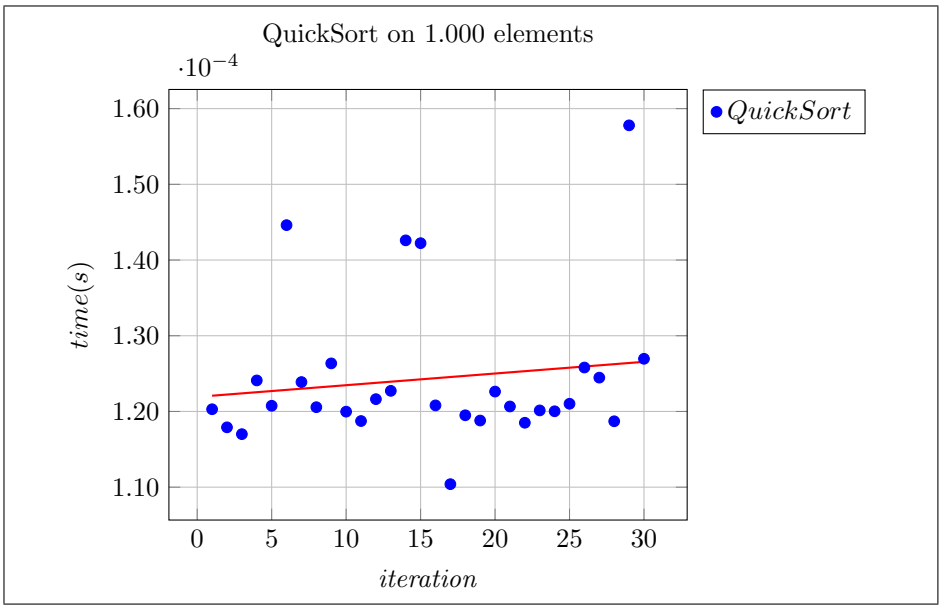

Figure E18: QuickSort scatter plot on 1.000 elements.

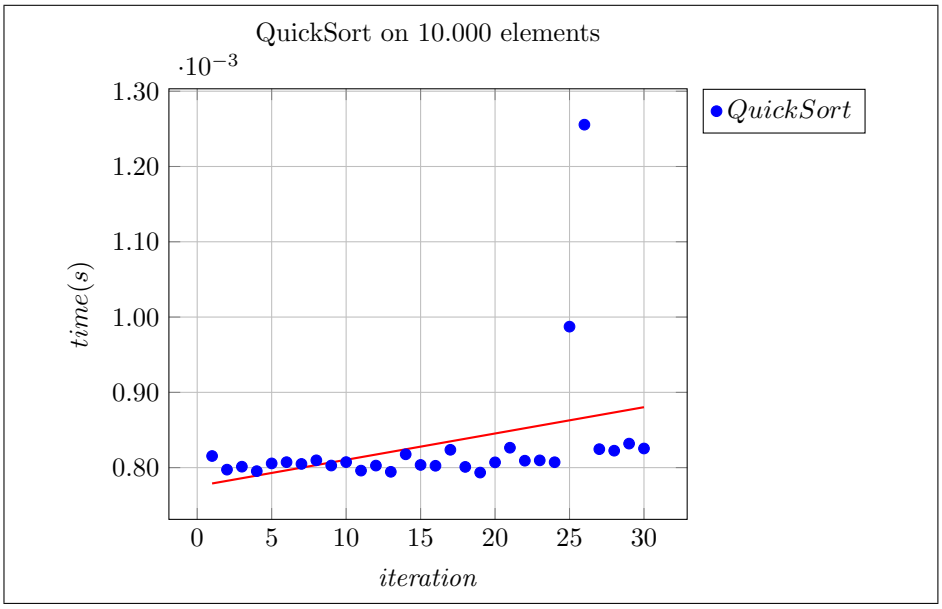

Figure E19: QuickSort scatter plot on 10.000 elements.

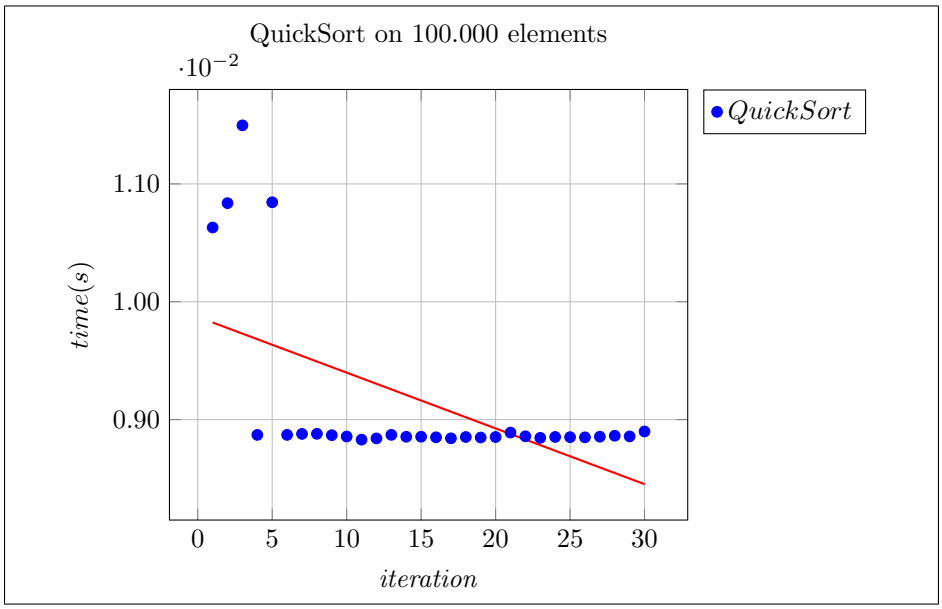

Figure E20: QuickSort scatter plot on 100.000 elements.

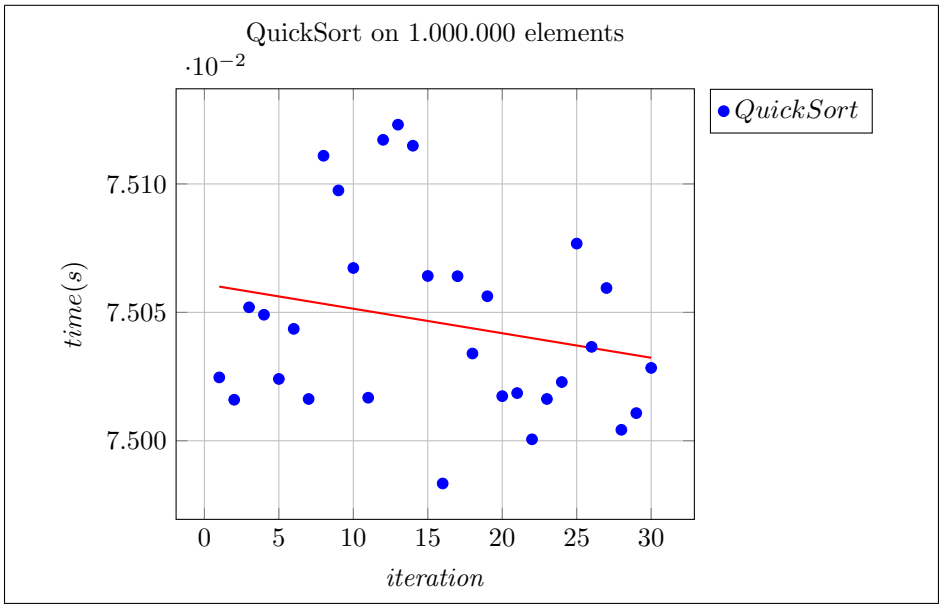

Figure E21: QuickSort scatter plot on 1.000.000 elements.

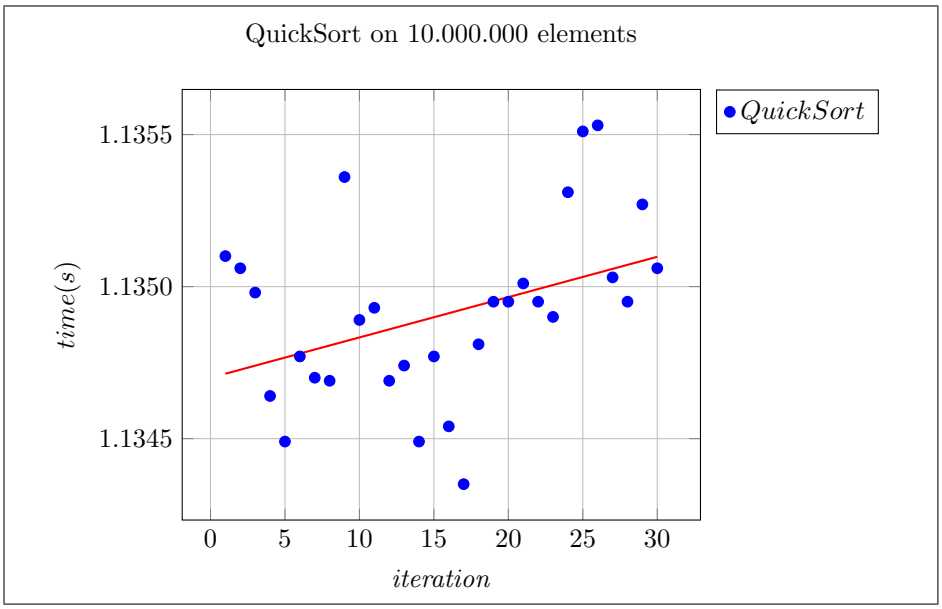

Figure E22: QuickSort scatter plot on 10.000.000 elements.

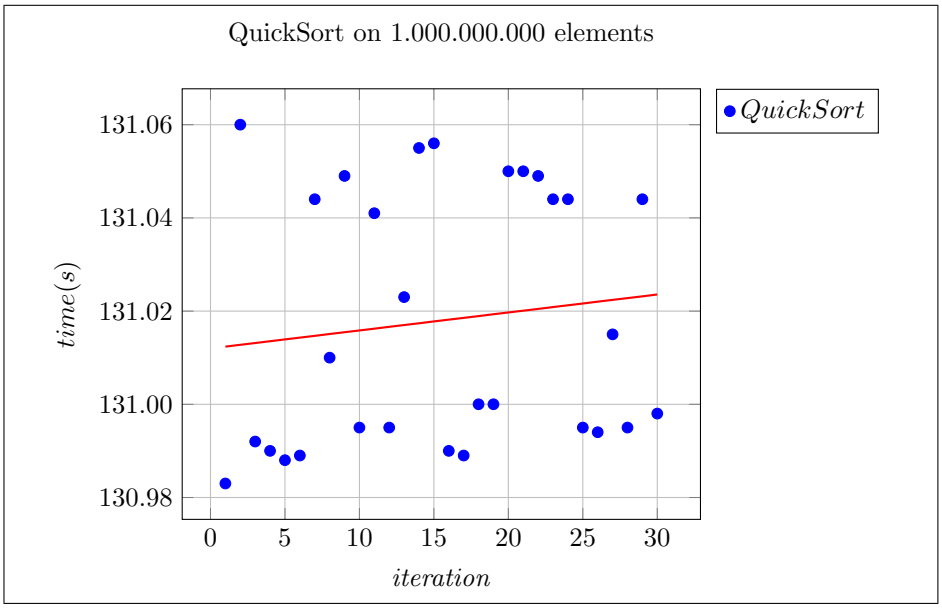

Figure E23: QuickSort scatter plot on 100.000.000 elements.

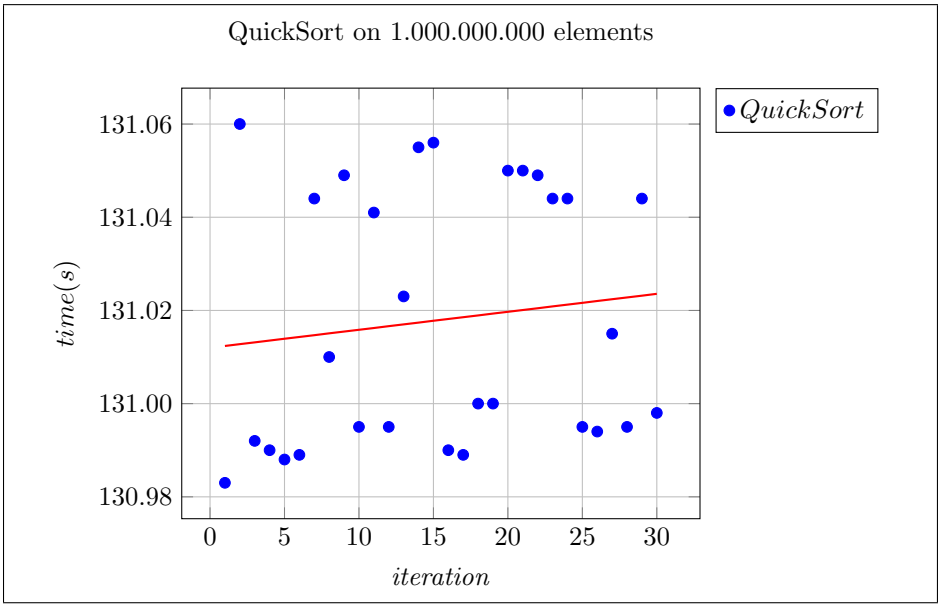

Figure E24: QuickSort scatter plot on 1.000.000.000 elements.

E.2 Encrypted Measurements

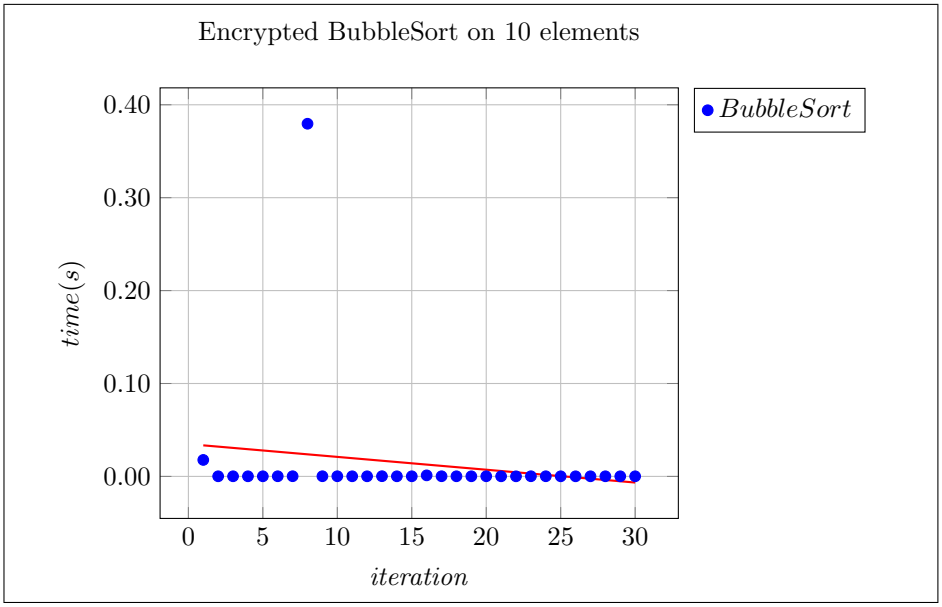

Figure E25: Encrypted BubbleSort scatter plot on 10 elements.

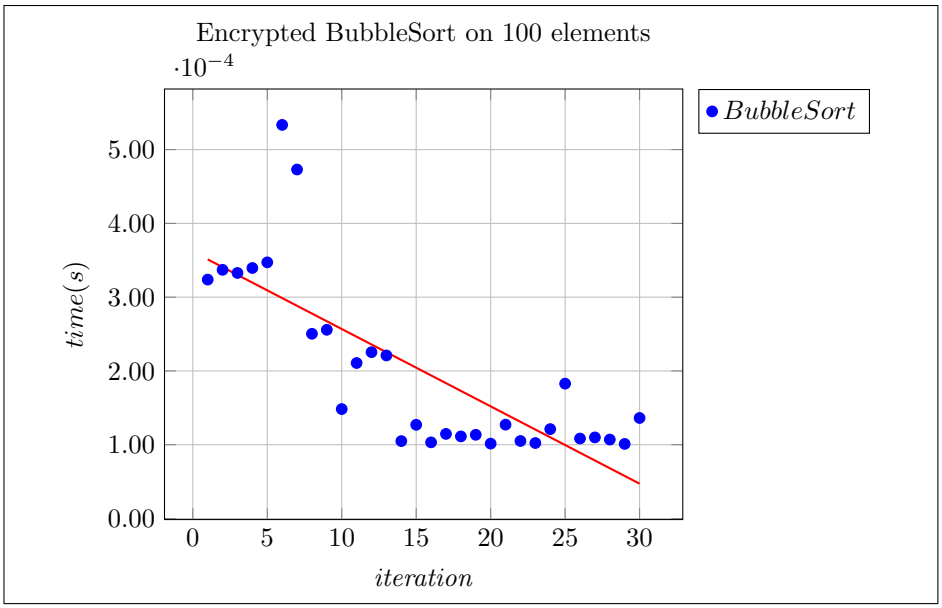

Figure E26: Encrypted BubbleSort scatter plot on 100 elements.

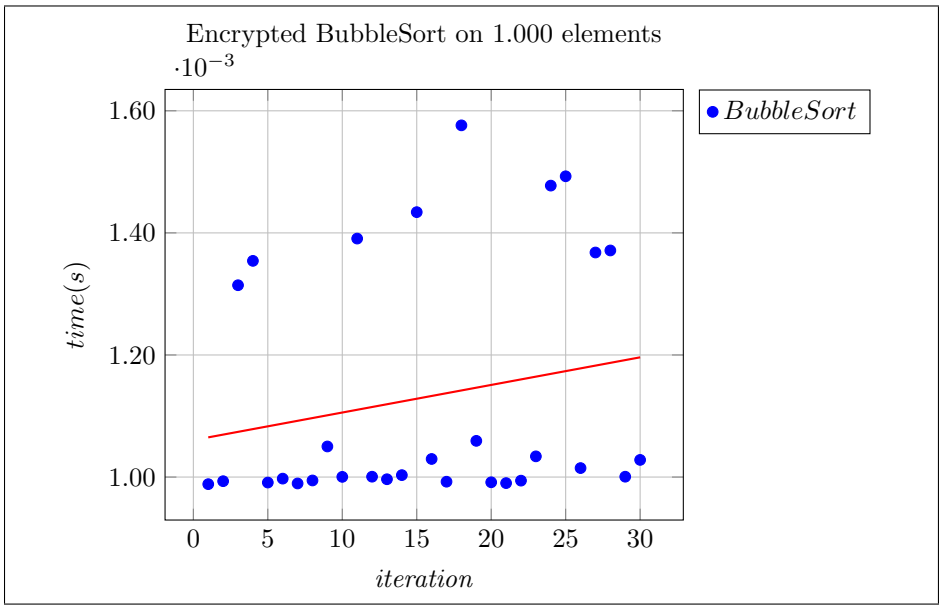

Figure E27: Encrypted BubbleSort scatter plot on 1.000 elements.

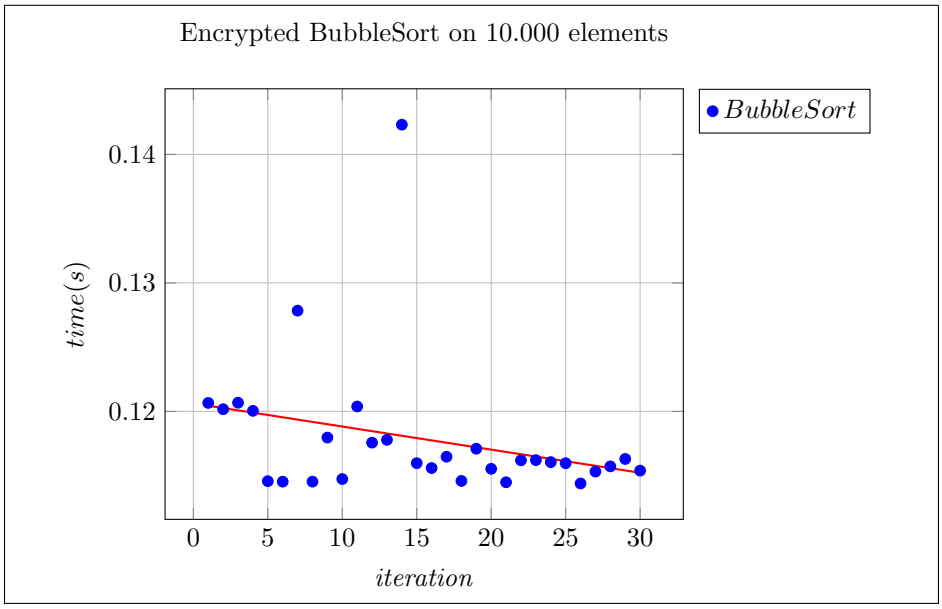

Figure E28: Encrypted BubbleSort scatter plot on 10.000 elements.

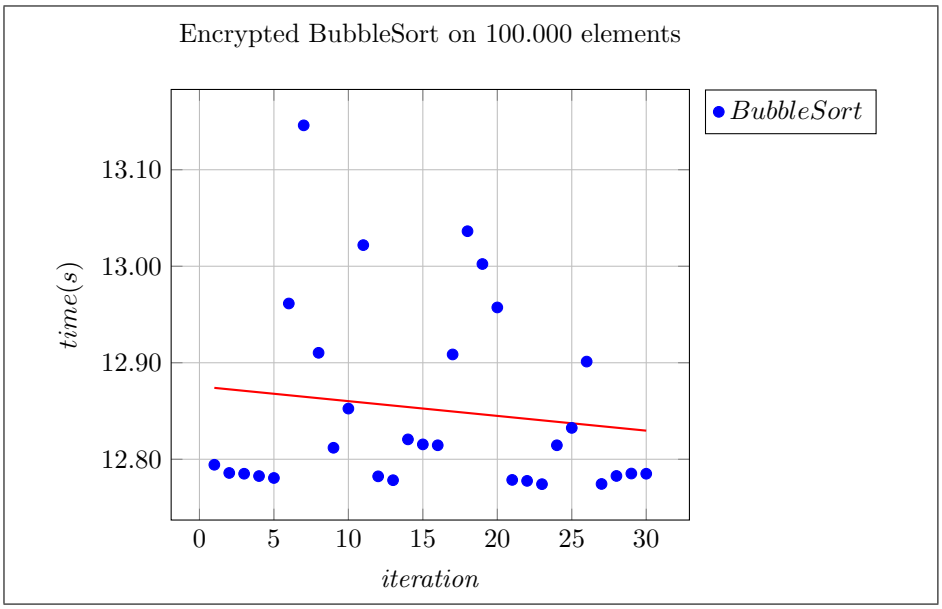

Figure E29: Encrypted BubbleSort scatter plot on 100.000 elements.

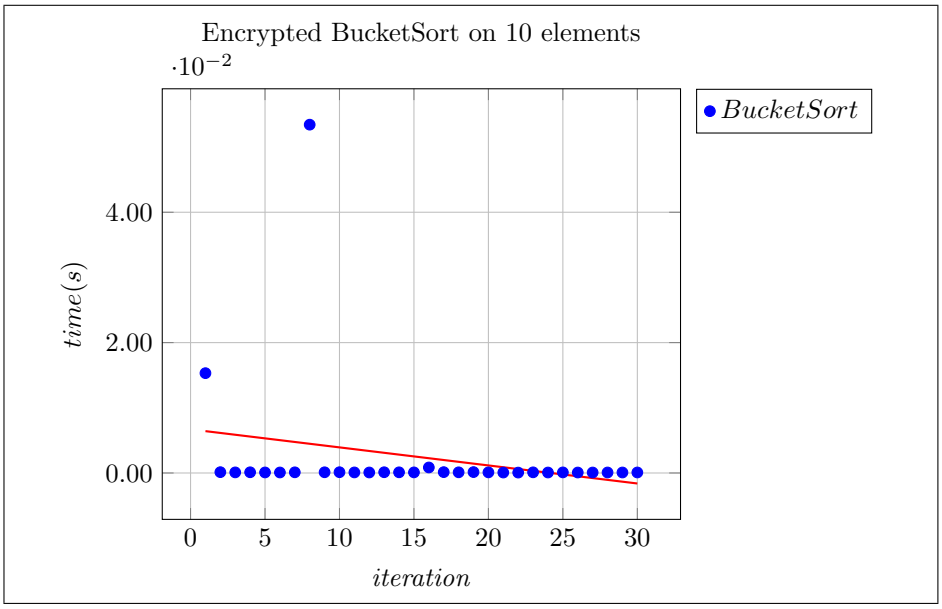

Figure E30: Encrypted BucketSort scatter plot on 10 elements.

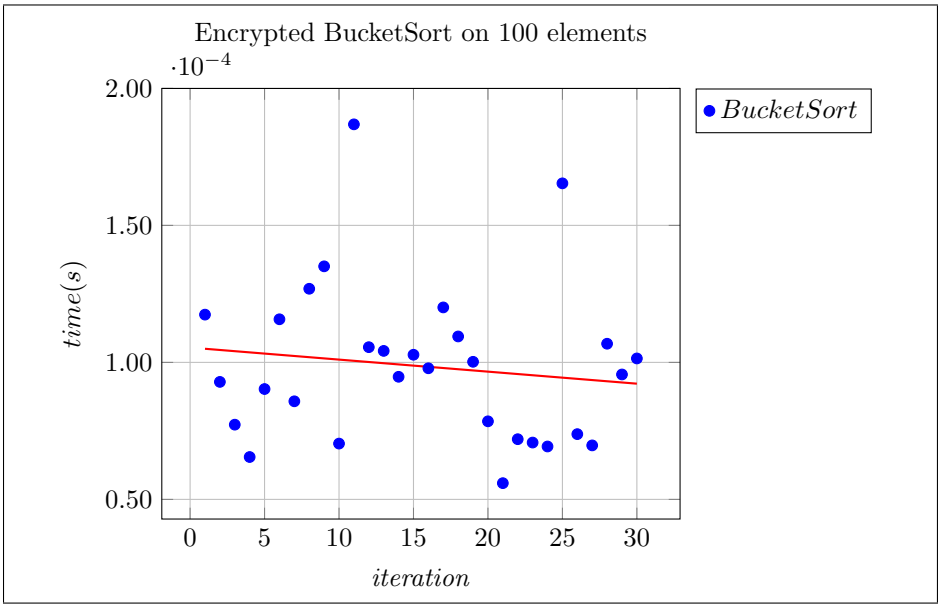

Figure E31: Encrypted BucketSort scatter plot on 100 elements.

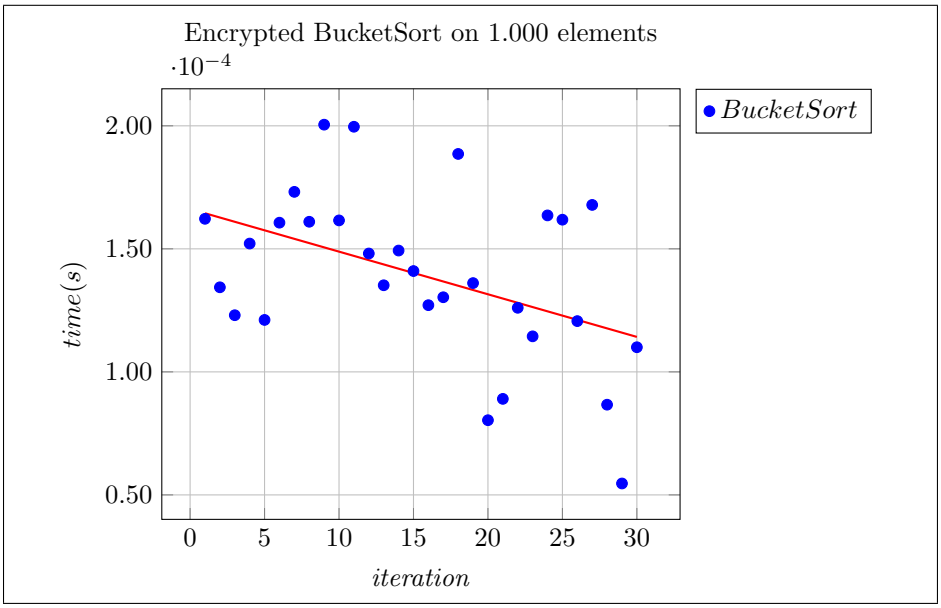

Figure E32: Encrypted BucketSort scatter plot on 1.000 elements.

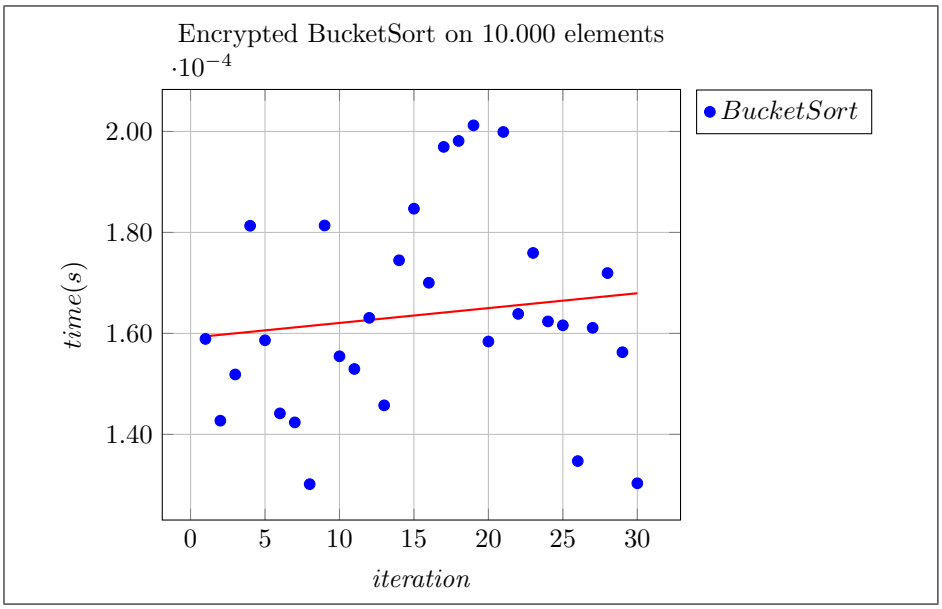

Figure E33: Encrypted BucketSort scatter plot on 10.000 elements.

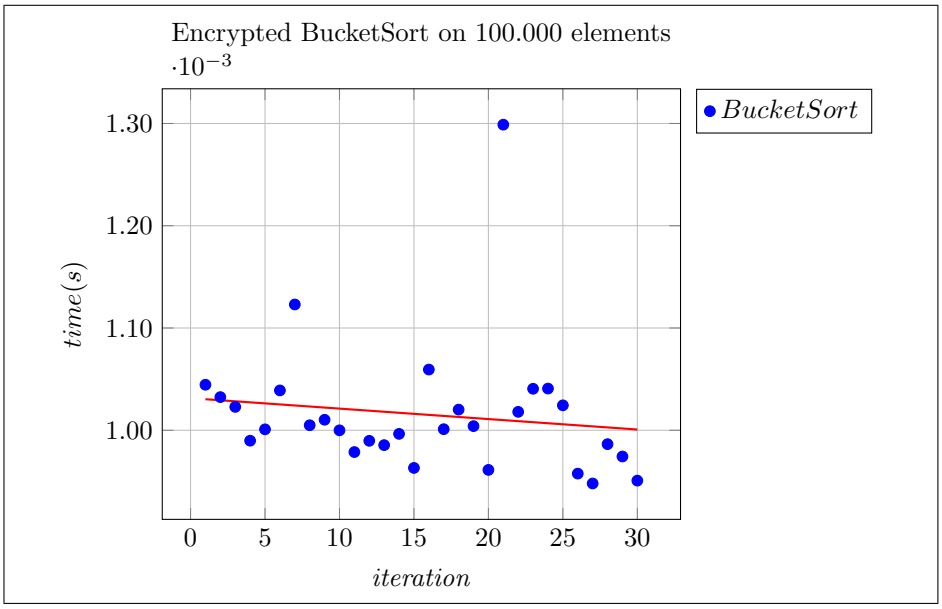

Figure E34: Encrypted BucketSort scatter plot on 100.000 elements.

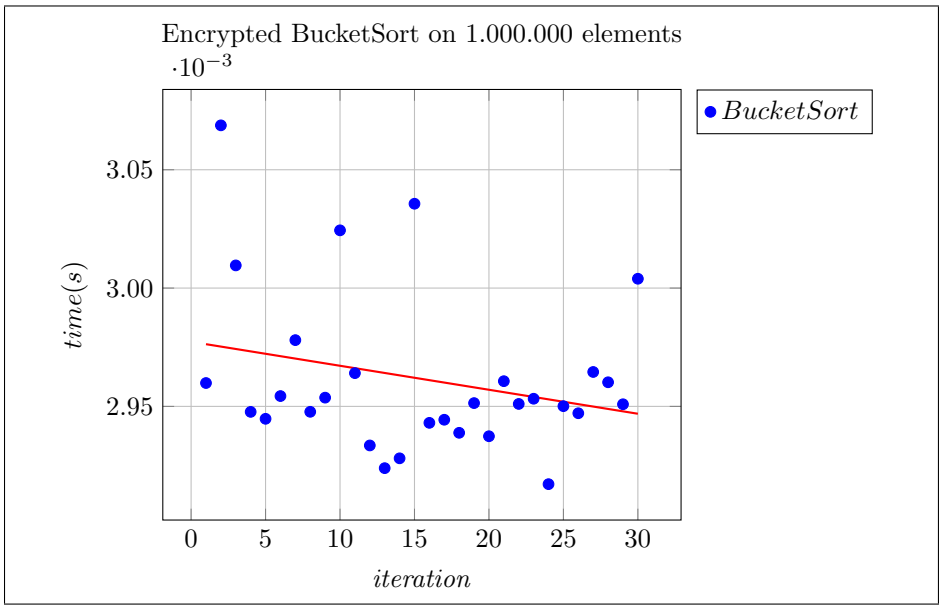

Figure E35: Encrypted BucketSort scatter plot on 1.000.000 elements.

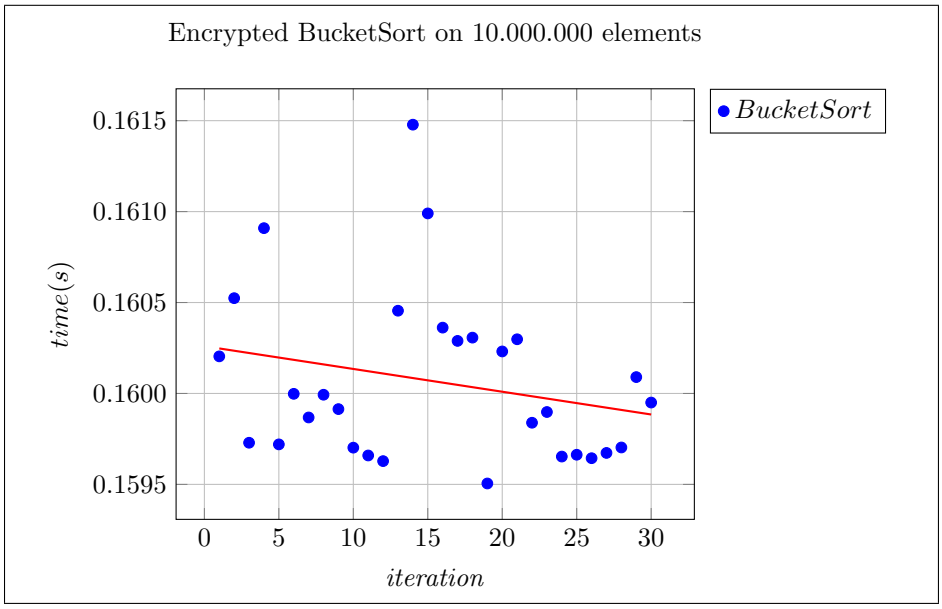

Figure E36: Encrypted BucketSort scatter plot on 10.000.000 elements.

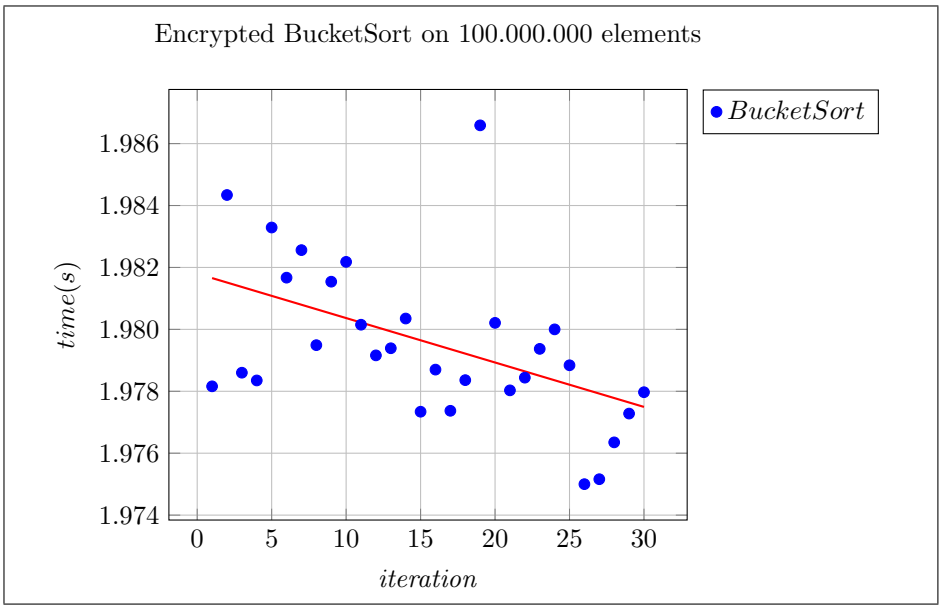

Figure E37: Encrypted BucketSort scatter plot on 100.000.000 elements.

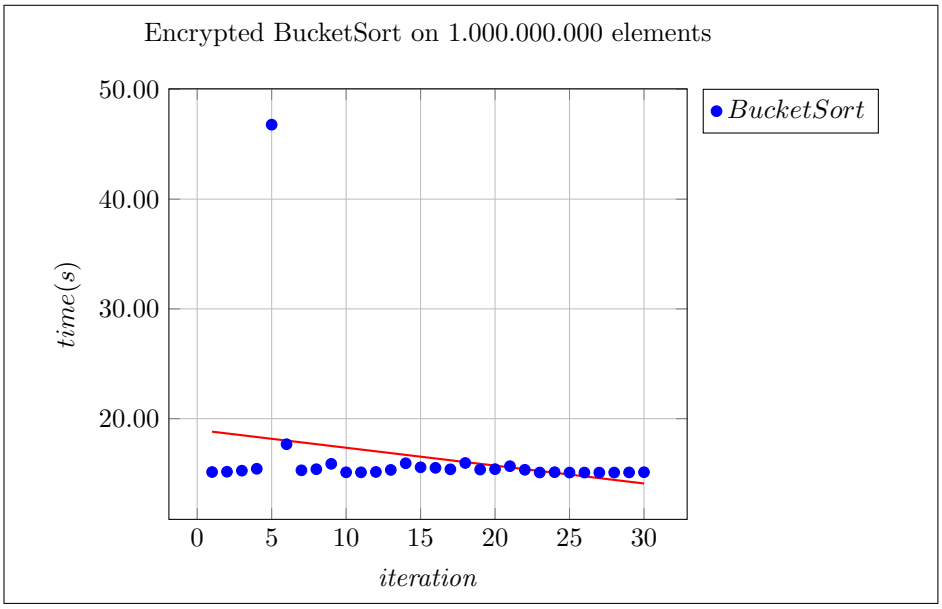

Figure E38: Encrypted BucketSort scatter plot on 1.000.000.000 elements.

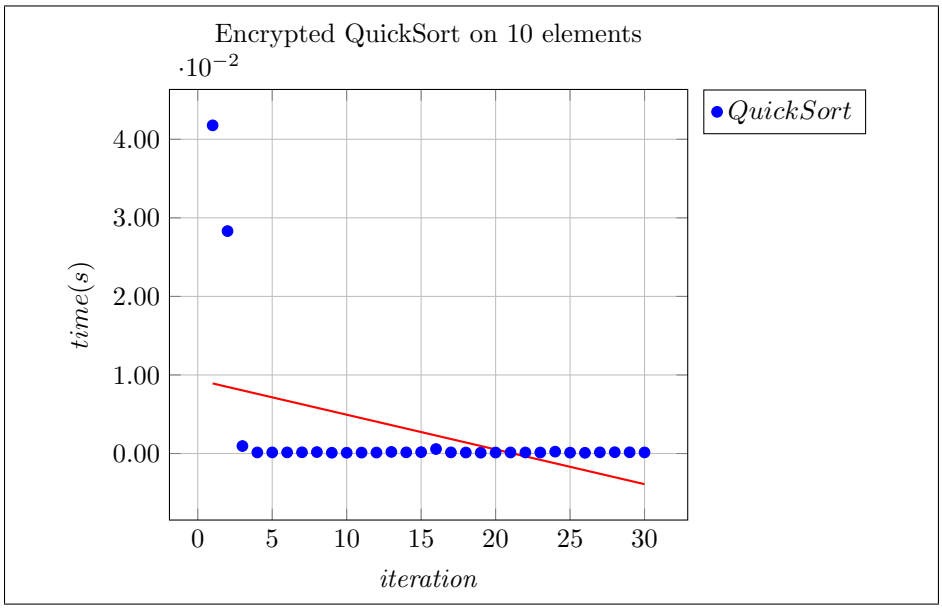

Figure E39: Encrypted QuickSort scatter plot on 10 elements.

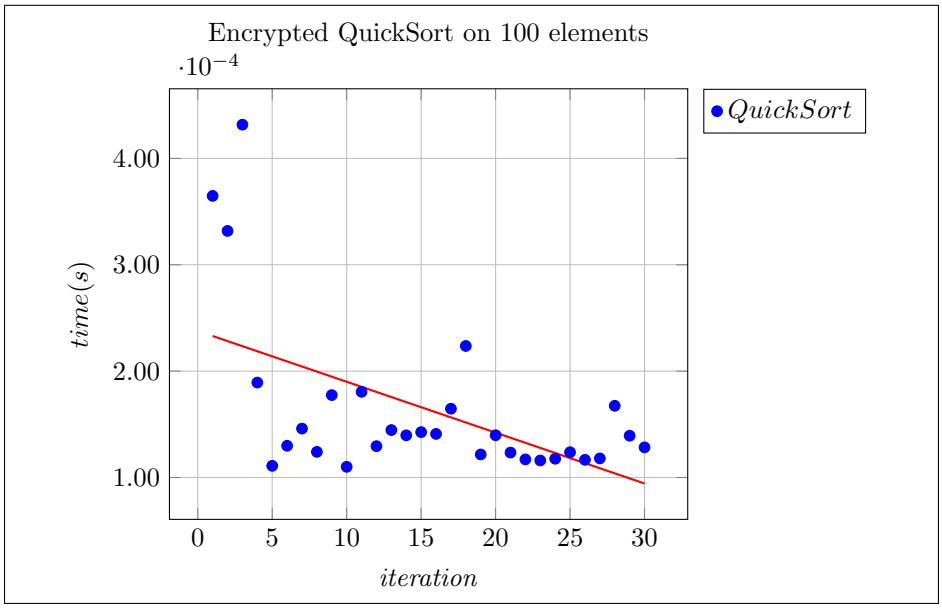

Figure E40: Encrypted QuickSort scatter plot on 100 elements.

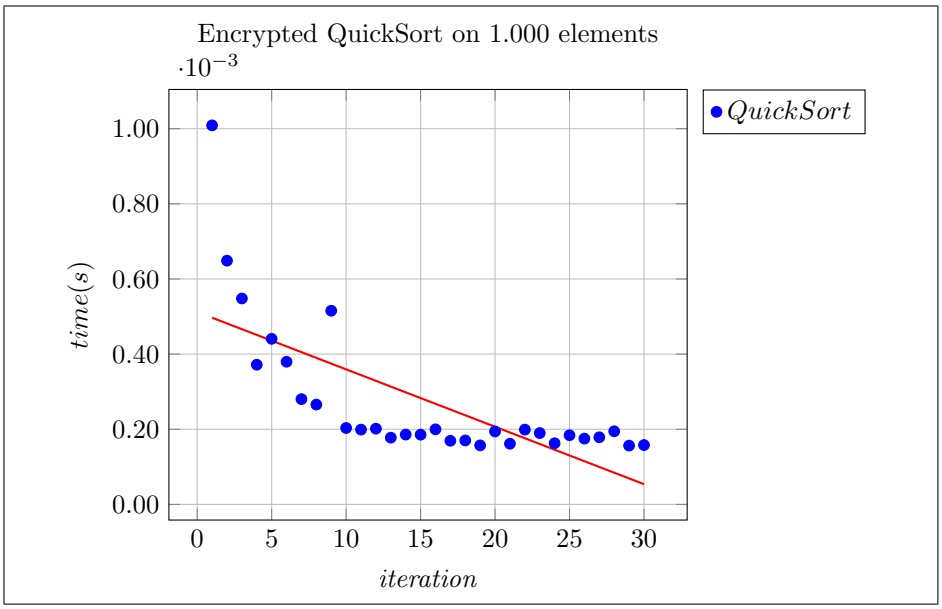

Figure E41: Encrypted QuickSort scatter plot on 1.000 elements.

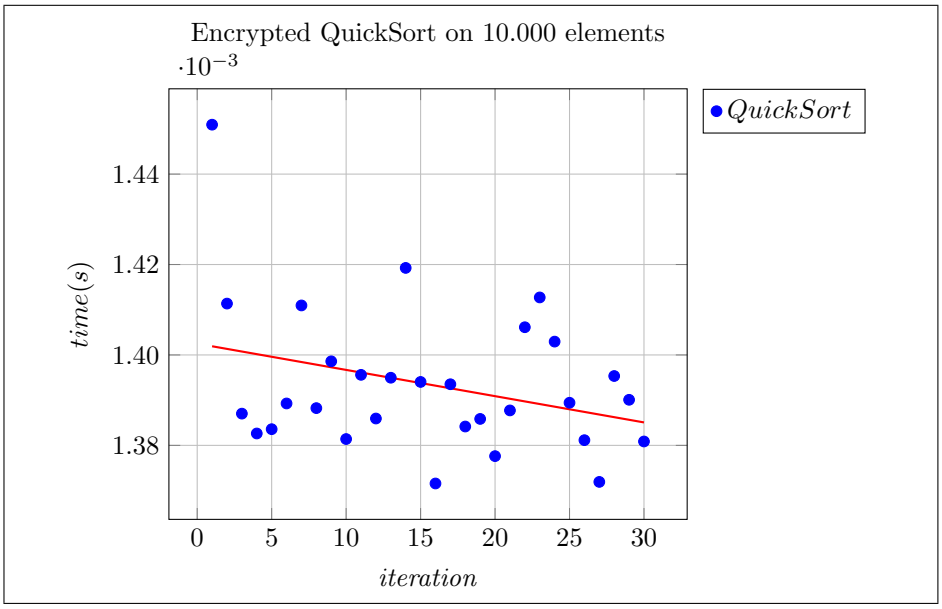

Figure E42: Encrypted QuickSort scatter plot on 10.000 elements.

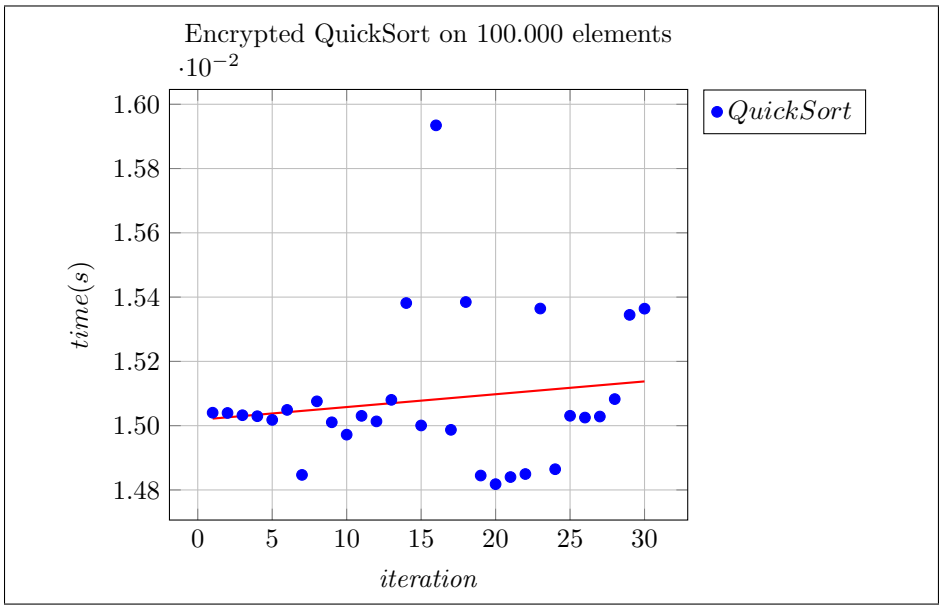

Figure E43: Encrypted QuickSort scatter plot on 100.000 elements.

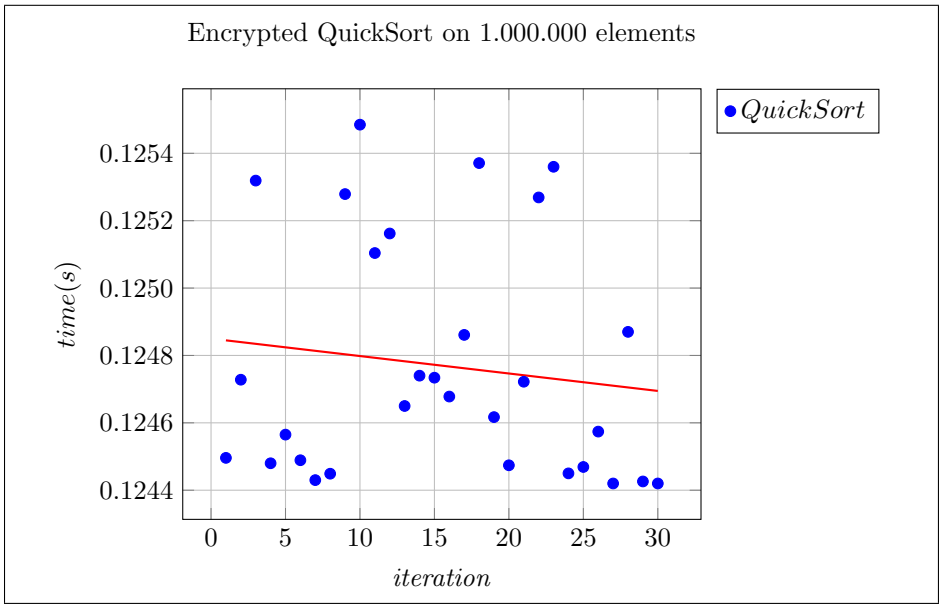

Figure E44: Encrypted QuickSort scatter plot on 1.000.000 elements.

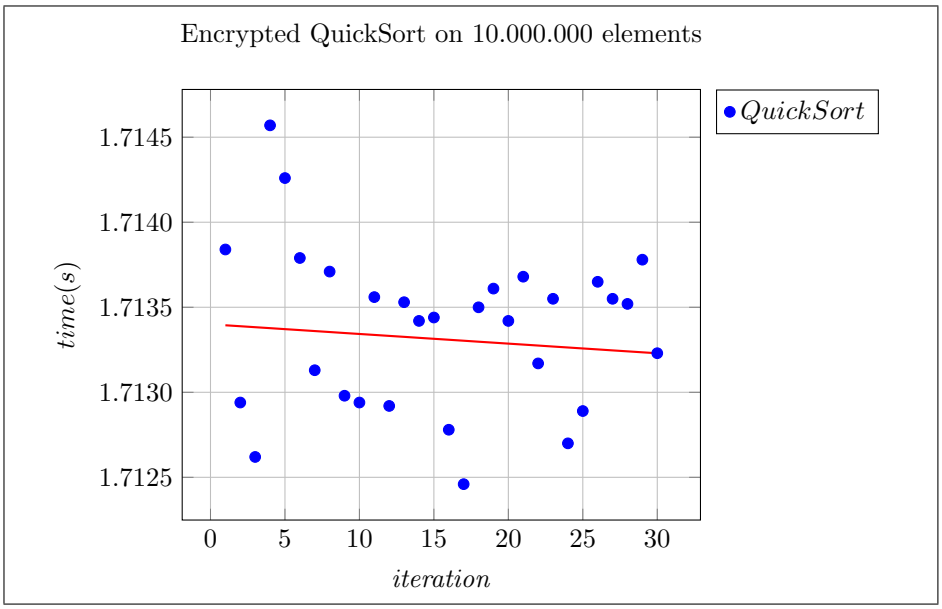

Figure E45: Encrypted QuickSort scatter plot on 10.000.000 elements.

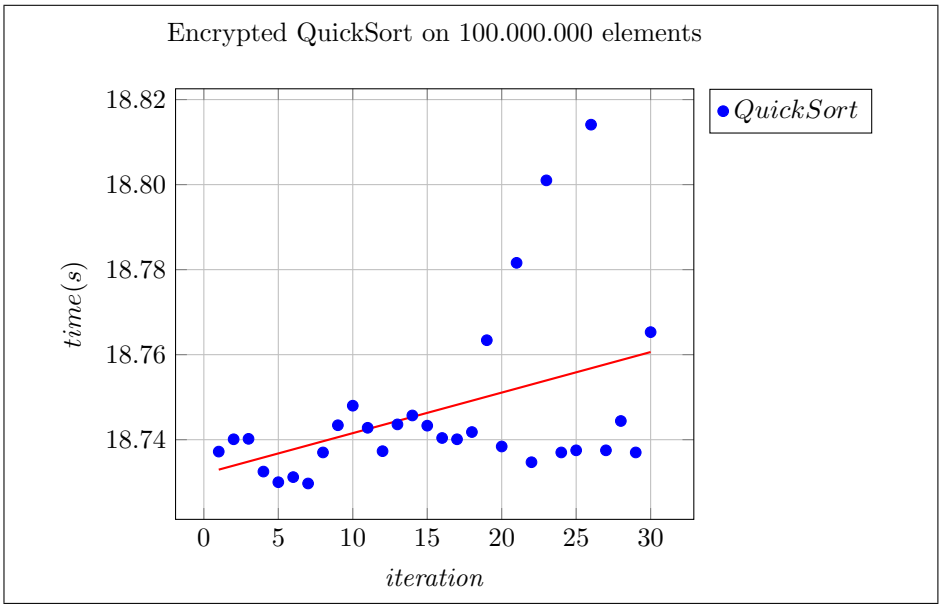

Figure E46: Encrypted QuickSort scatter plot on 100.000.000 elements.

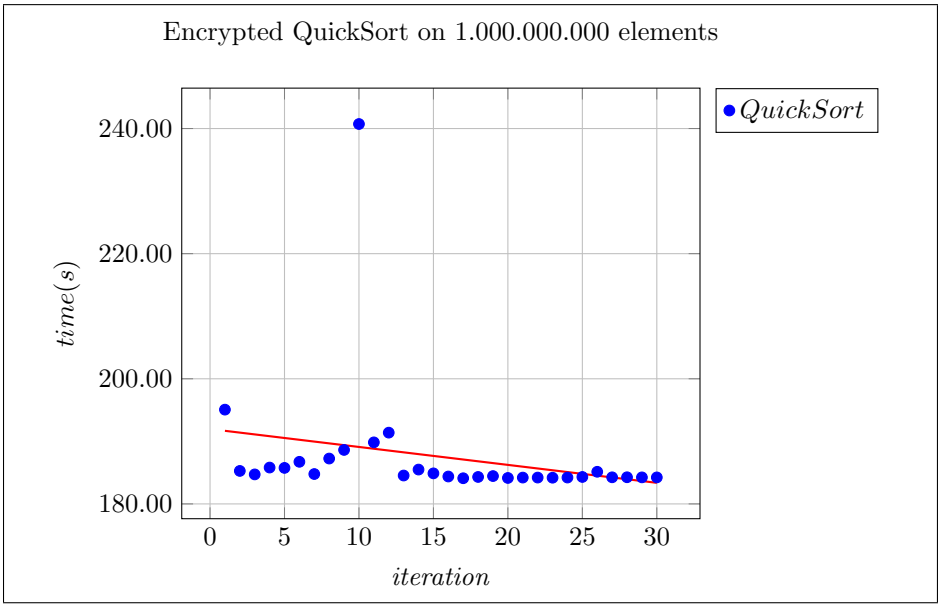

Figure E47: Encrypted QuickSort scatter plot on 1.000.000.000 elements.

E.3 Virtualized Measurements

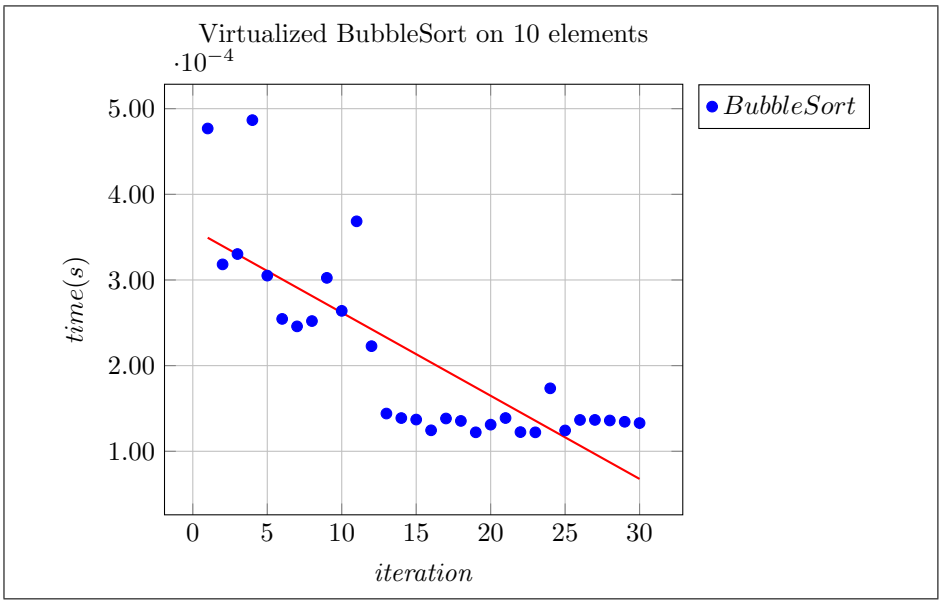

Figure E48: Virtualized BubbleSort scatter plot on 10 elements.

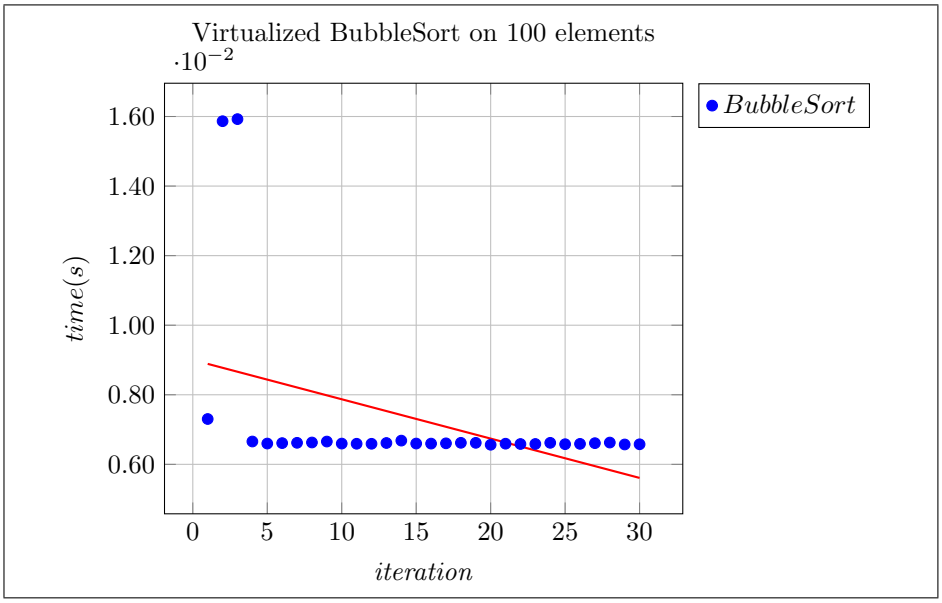

Figure E49: Virtualized BubbleSort scatter plot on 100 elements.

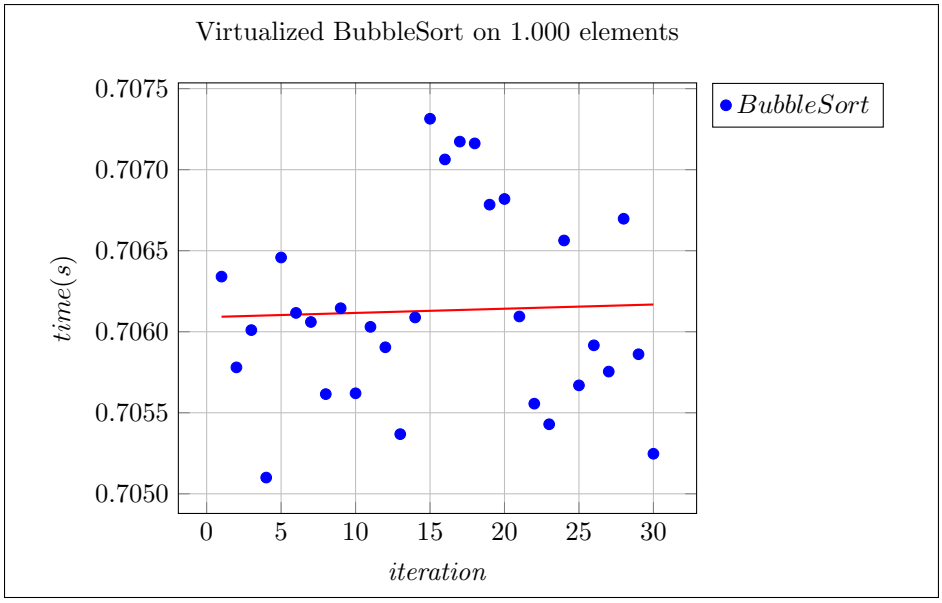

Figure E50: Virtualized BubbleSort scatter plot on 1.000 elements.

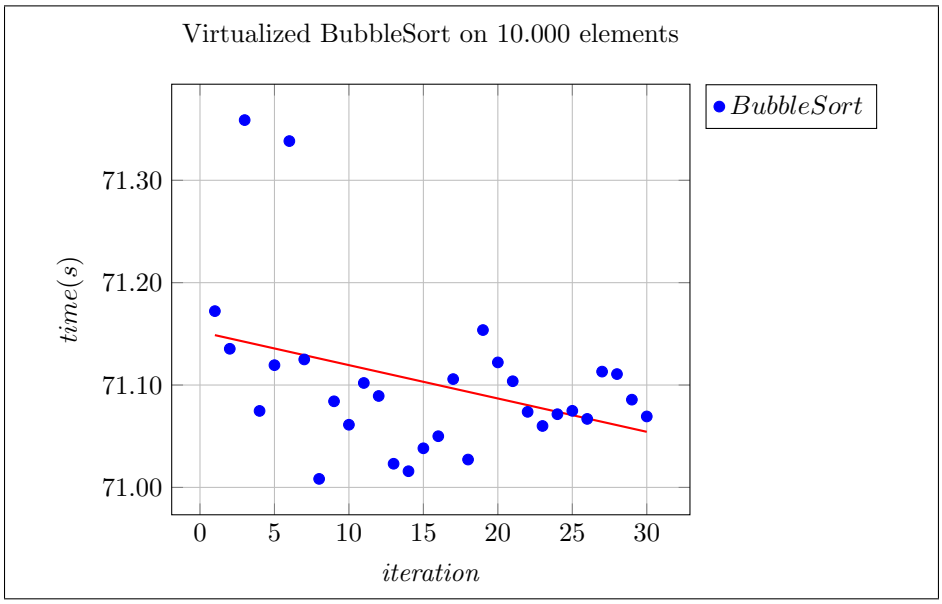

Figure E51: Virtualized BubbleSort scatter plot on 10.000 elements.

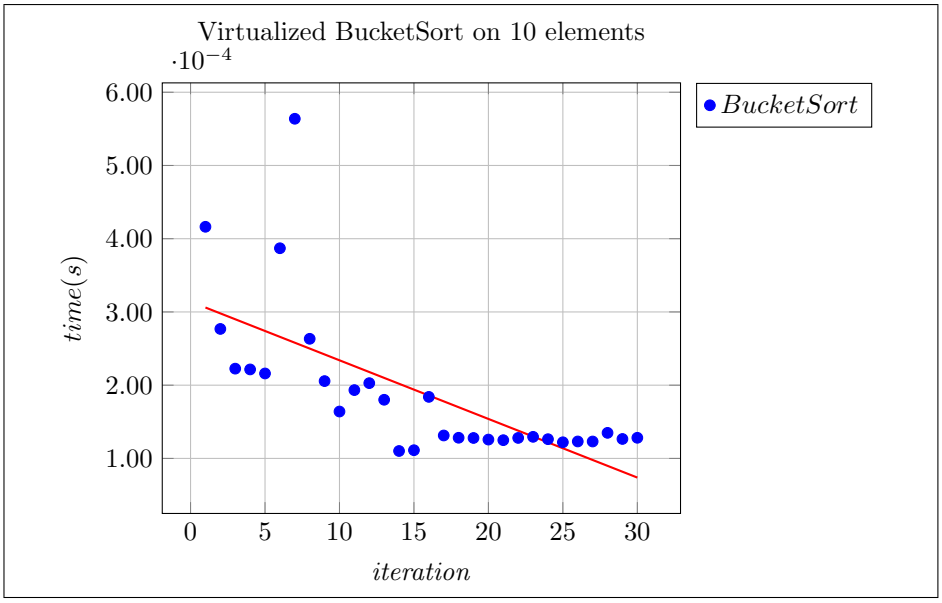

Figure E52: Virtualized BucketSort scatter plot on 10 elements.

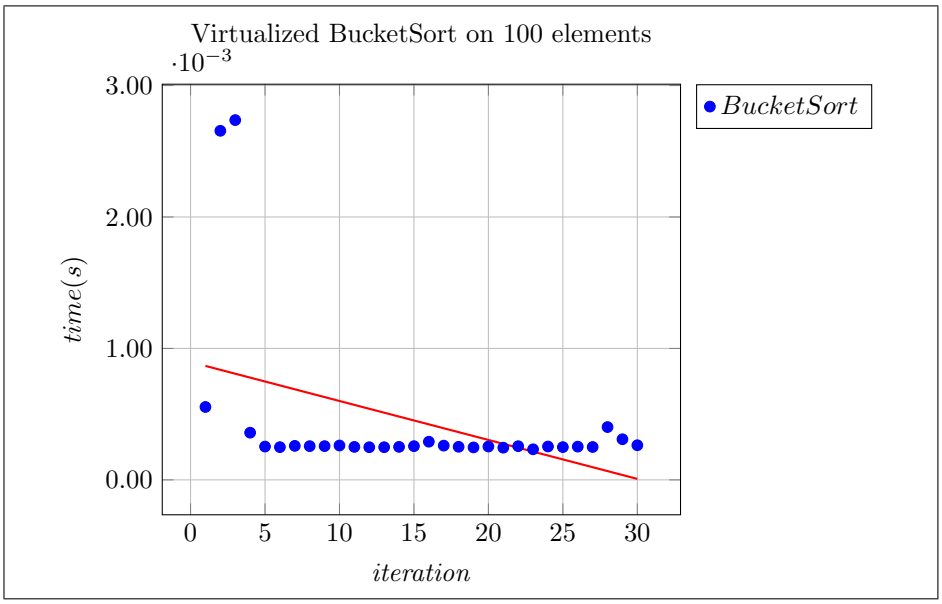

Figure E53: Virtualized BucketSort scatter plot on 100 elements.

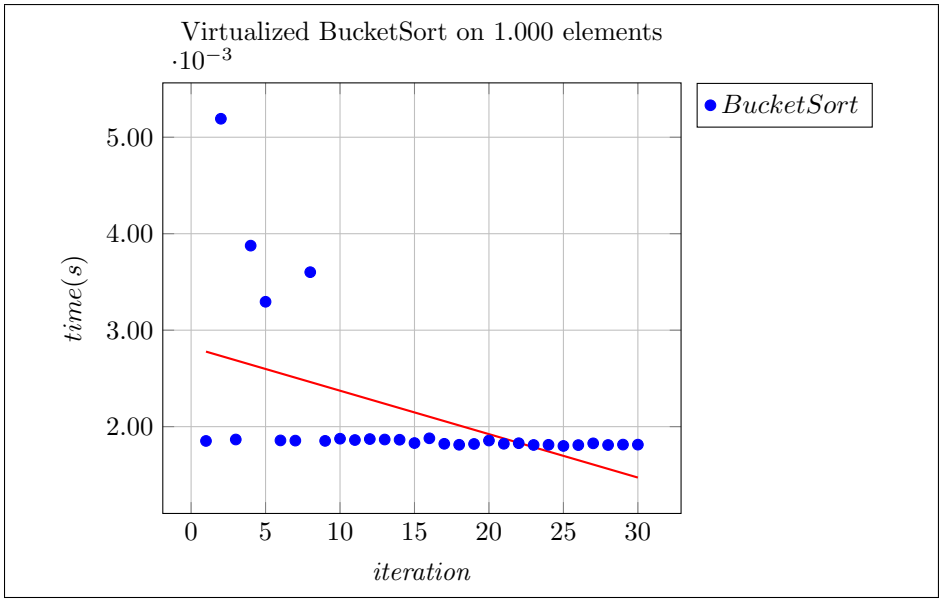

Figure E54: Virtualized BucketSort scatter plot on 1.000 elements.

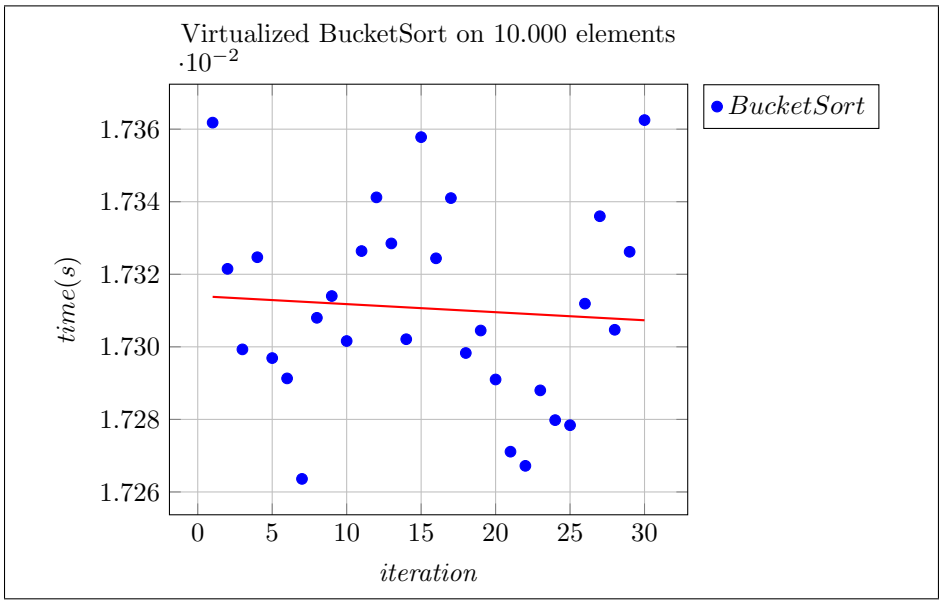

Figure E55: Virtualized BucketSort scatter plot on 10.000 elements.

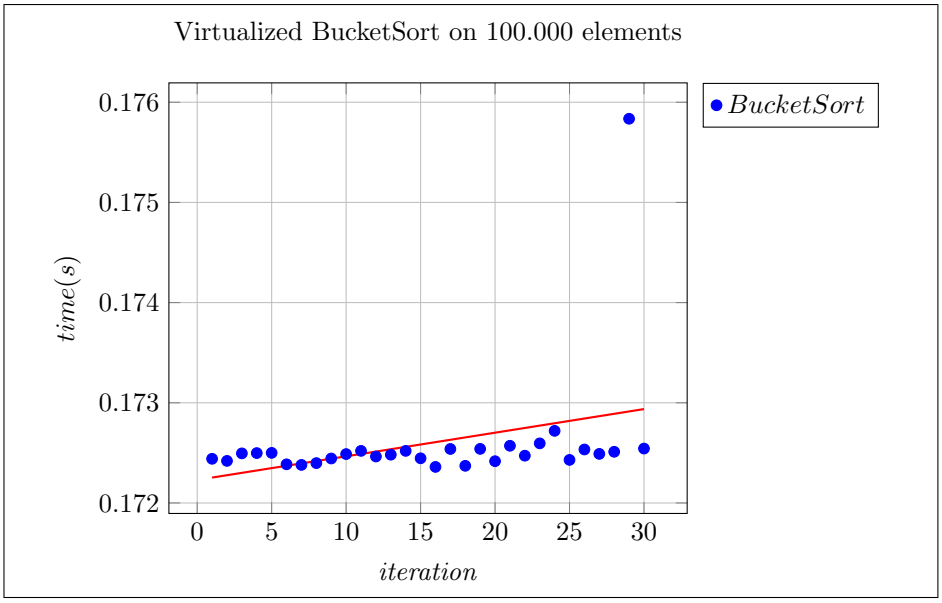

Figure E56: Virtualized BucketSort scatter plot on 100.000 elements.

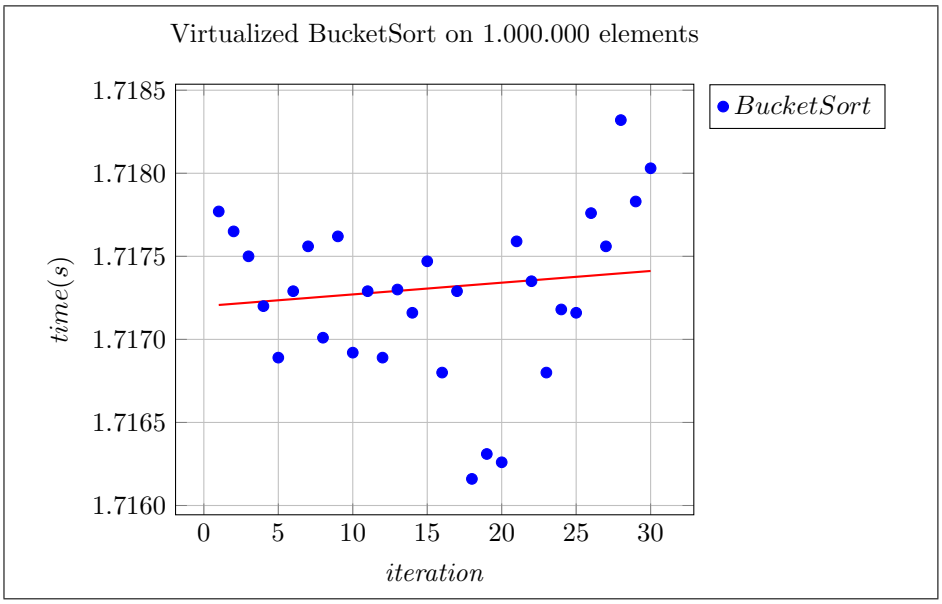

Figure E57: Virtualized BucketSort scatter plot on 1.000.000 elements.

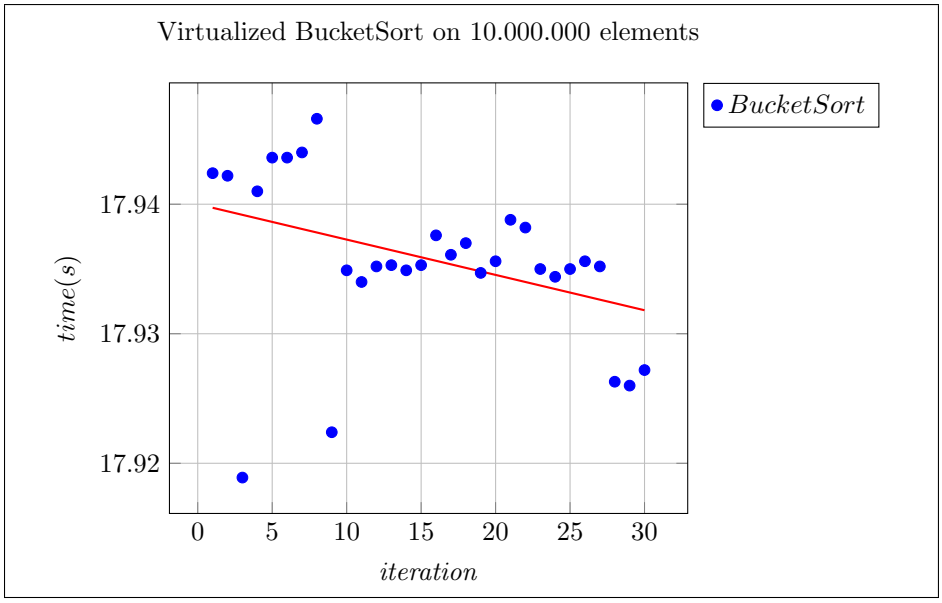

Figure E58: Virtualized BucketSort scatter plot on 10.000.000 elements.

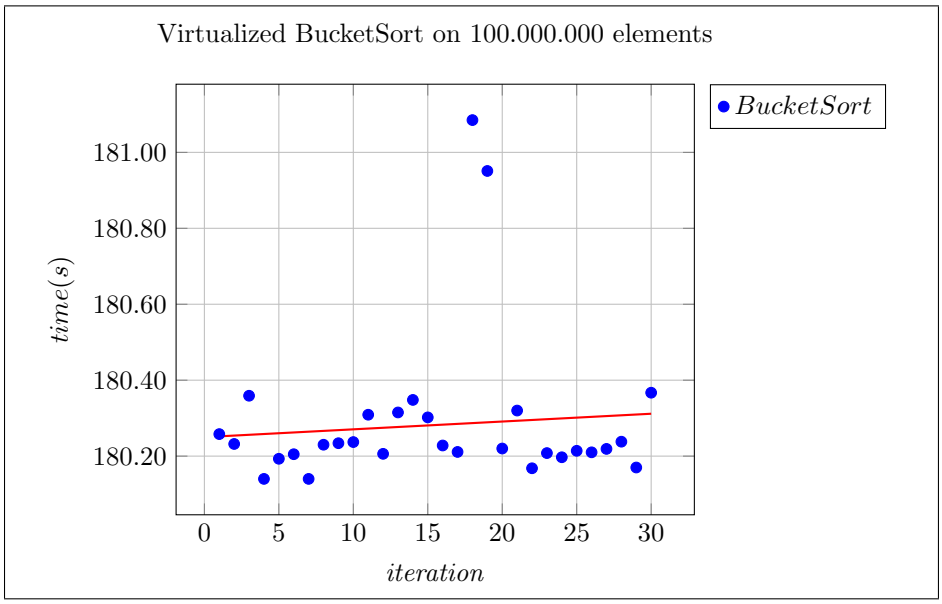

Figure E59: Virtualized BucketSort scatter plot on 100.000.000 elements.

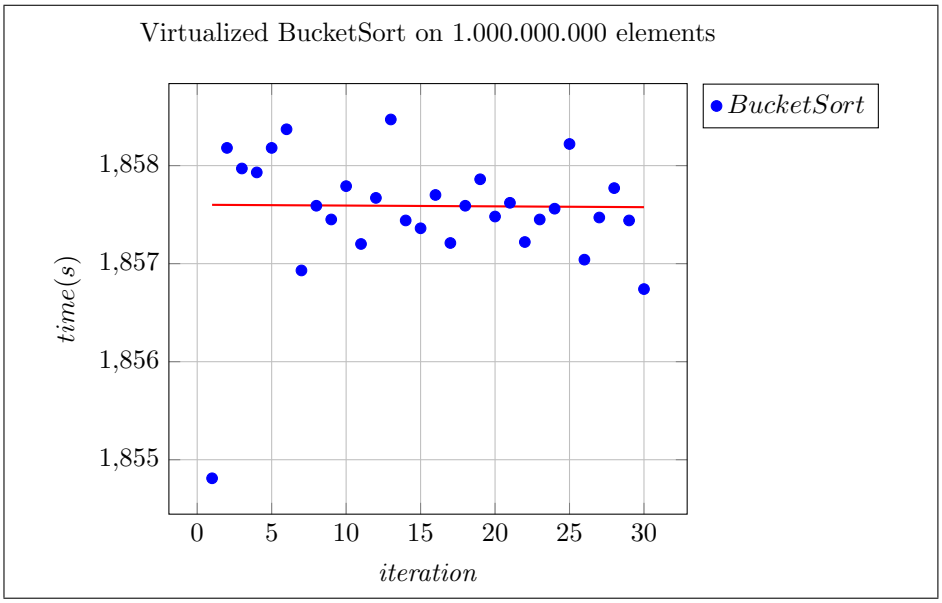

Figure E60: Virtualized BucketSort scatter plot on 1.000.000.000 elements.

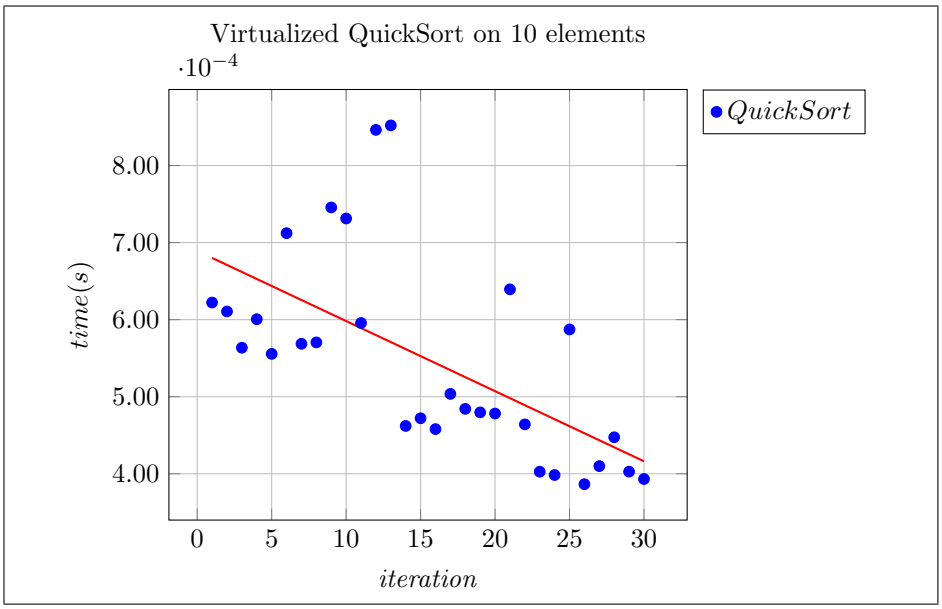

Figure E61: Virtualized QuickSort scatter plot on 10 elements.

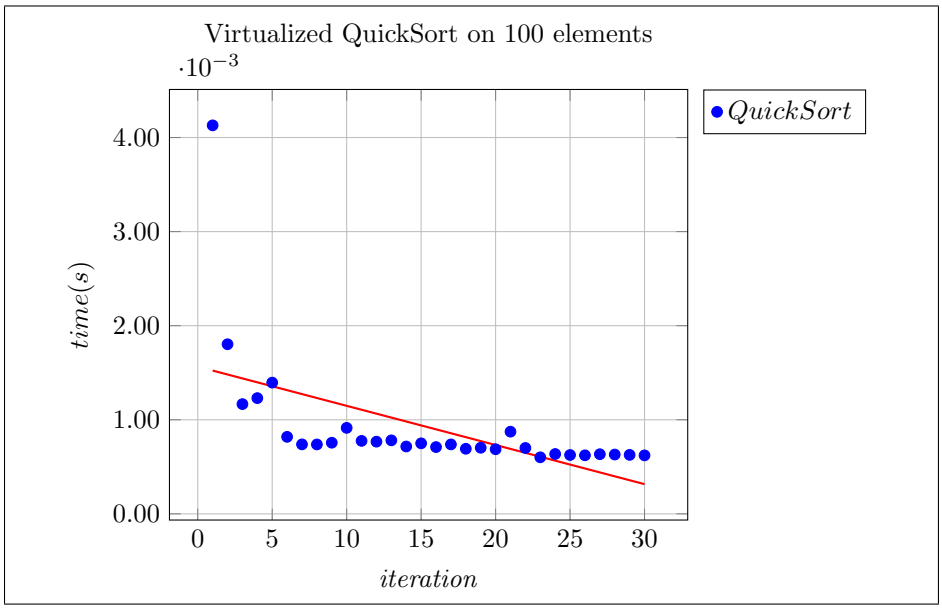

Figure E62: Virtualized QuickSort scatter plot on 100 elements.

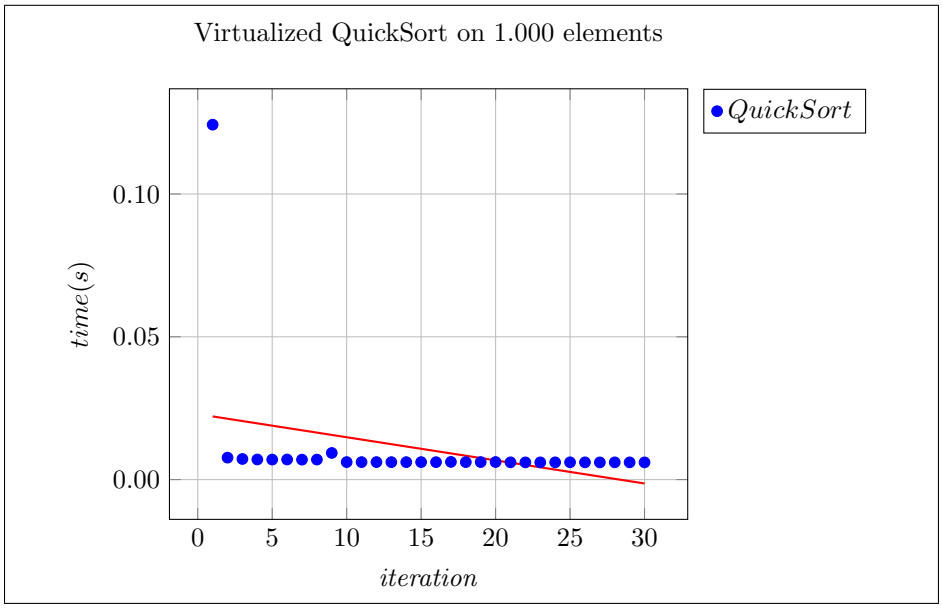

Figure E63: Virtualized QuickSort scatter plot on 1.000 elements.

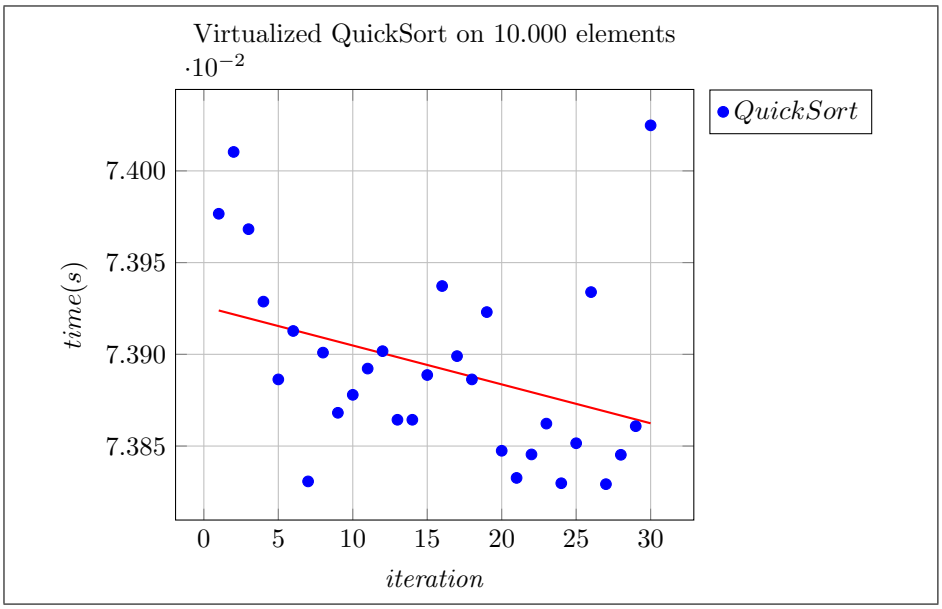

Figure E64: Virtualized QuickSort scatter plot on 10.000 elements.

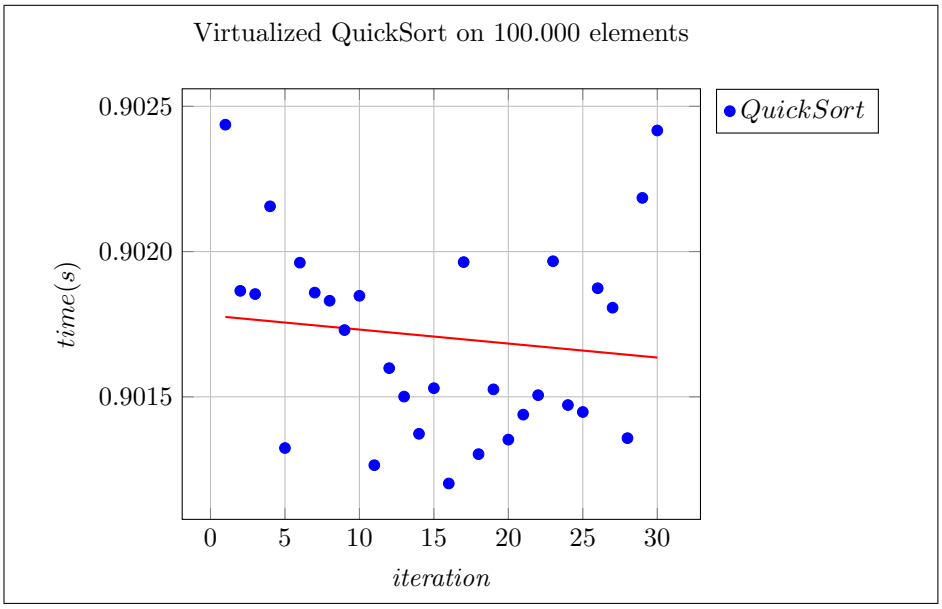

Figure E65: Virtualized QuickSort scatter plot on 100.000 elements.

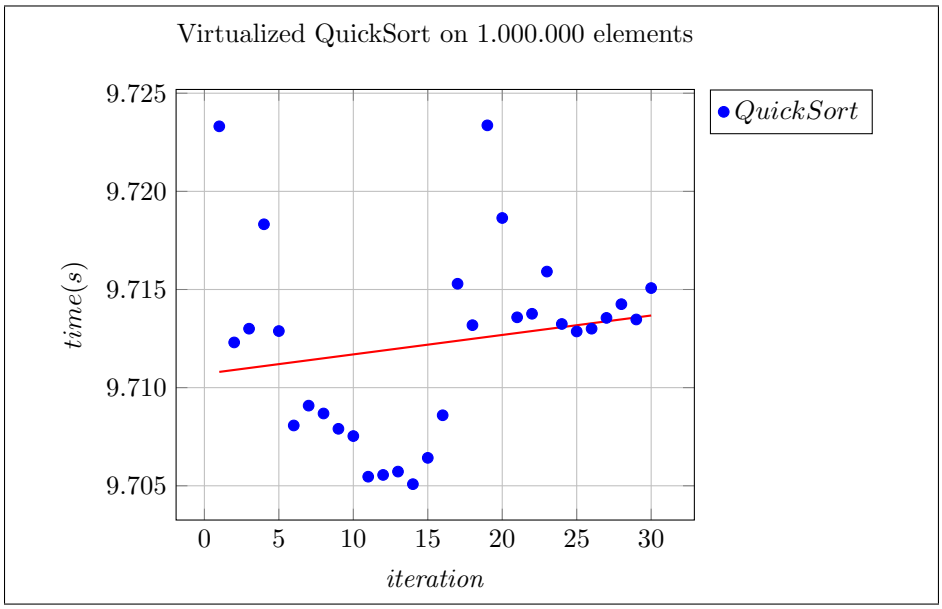

Figure E66: Virtualized QuickSort scatter plot on 1.000.000 elements.

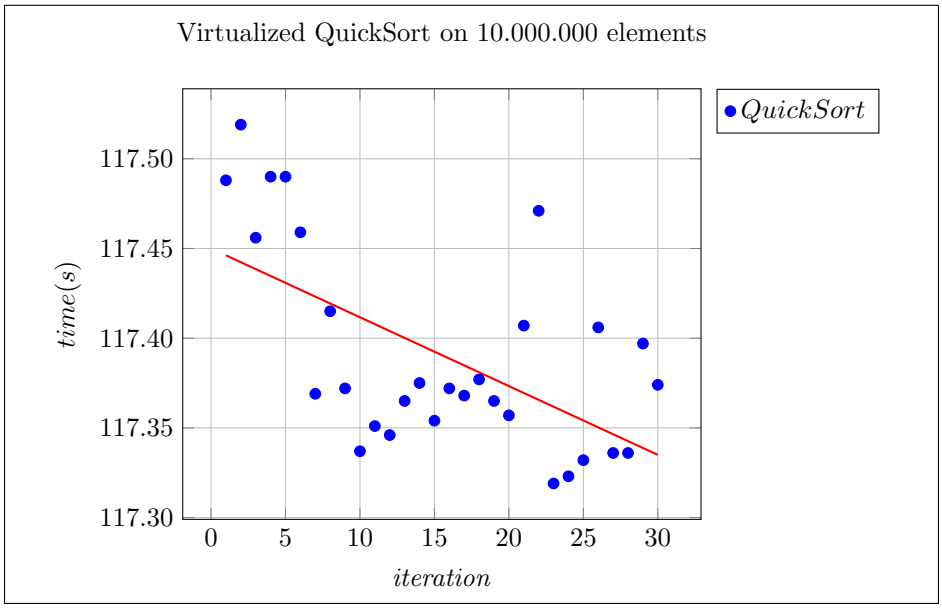

Figure E67: Virtualized QuickSort scatter plot on 10.000.000 elements.

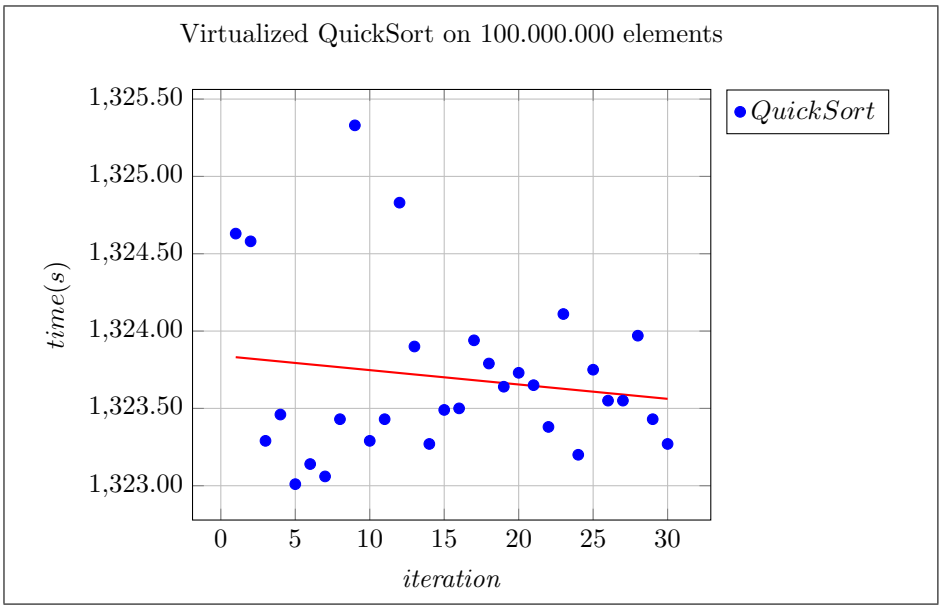

Figure E68: Virtualized QuickSort scatter plot on 100.000.000 elements.

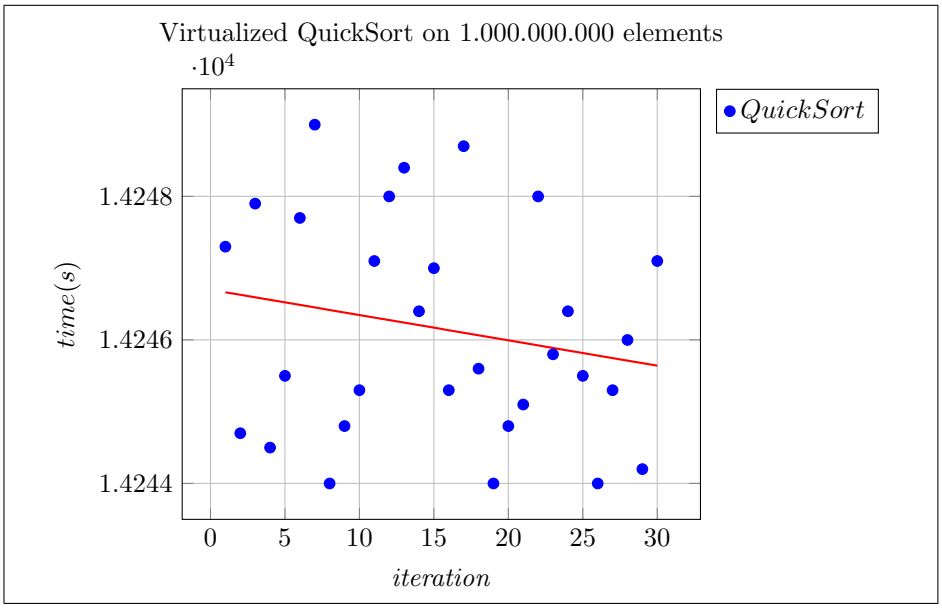

Figure E69: Virtualized QuickSort scatter plot on 1.000.000.000 elements.

E.4 Threading Reference

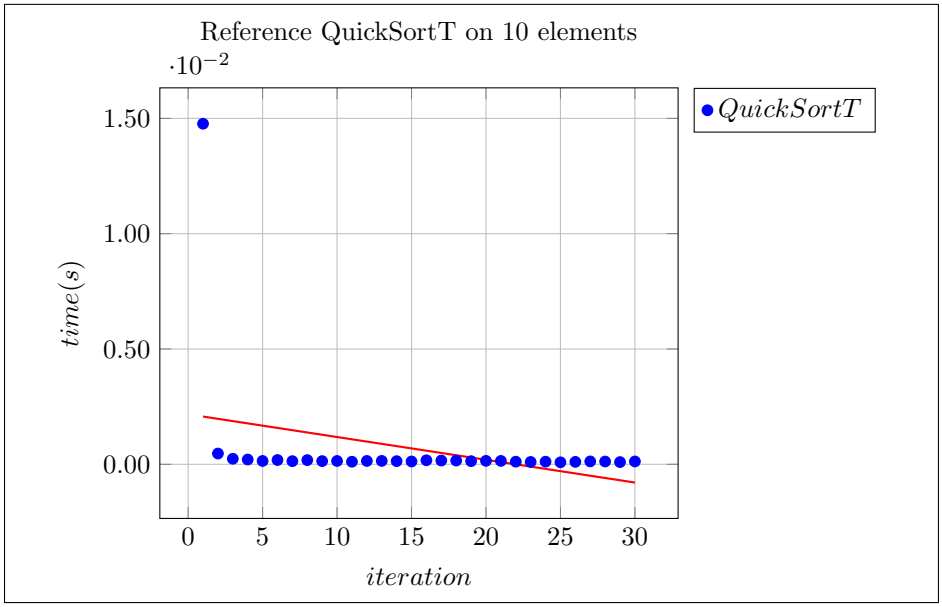

Figure E70: QuickSortT scatter plot on 10 elements.

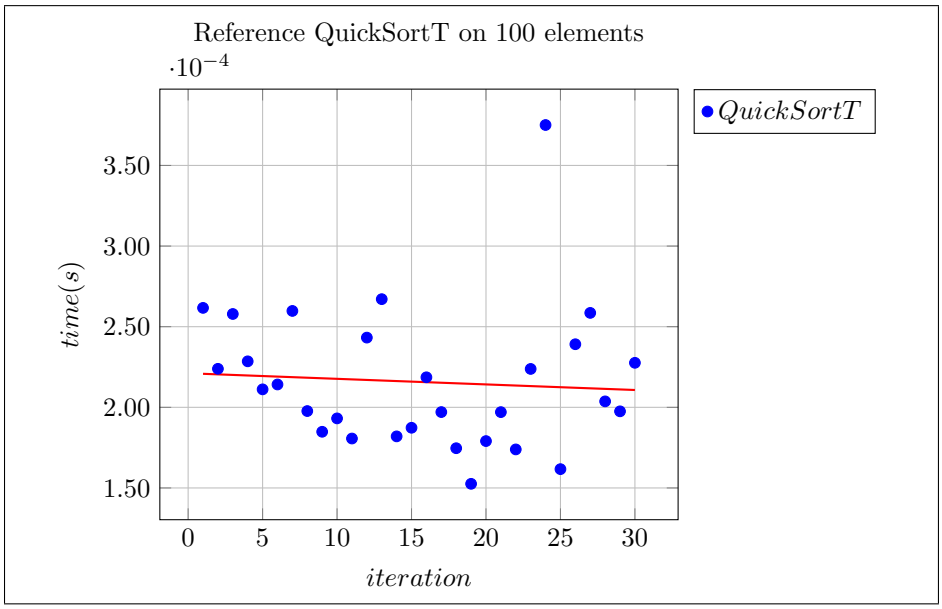

Figure E71: QuickSortT scatter plot on 100 elements.

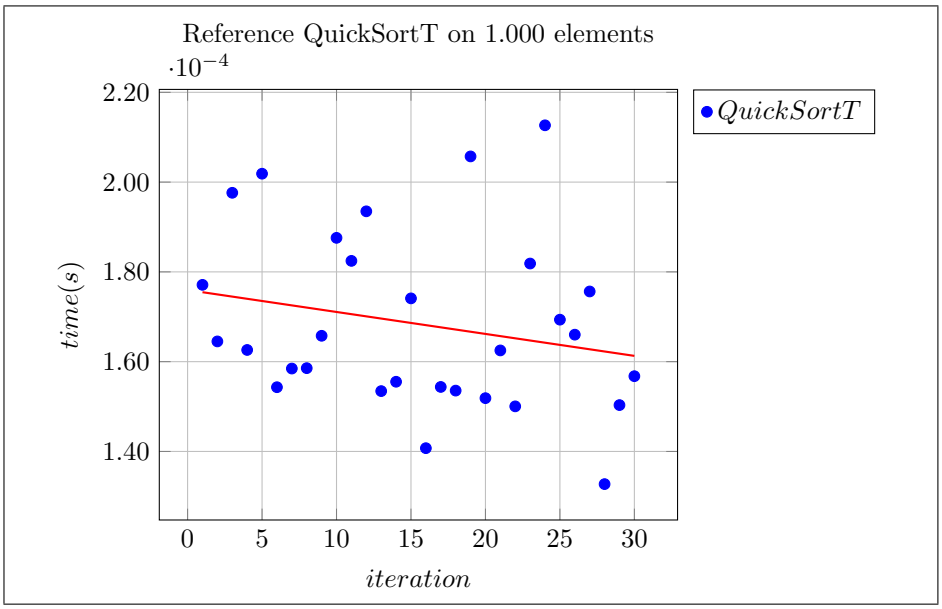

Figure E72: QuickSortT scatter plot on 1.000 elements.

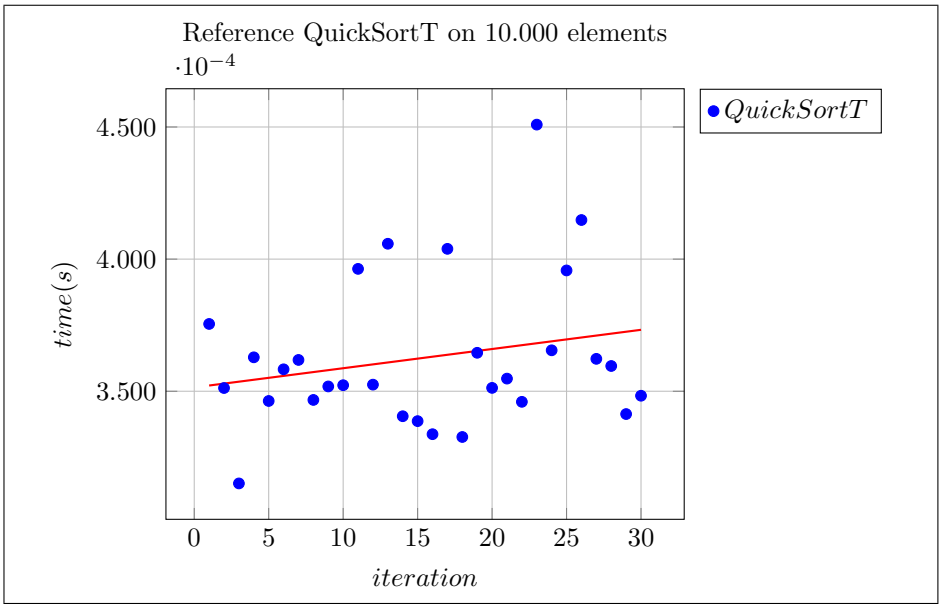

Figure E73: QuickSortT scatter plot on 10.000 elements.

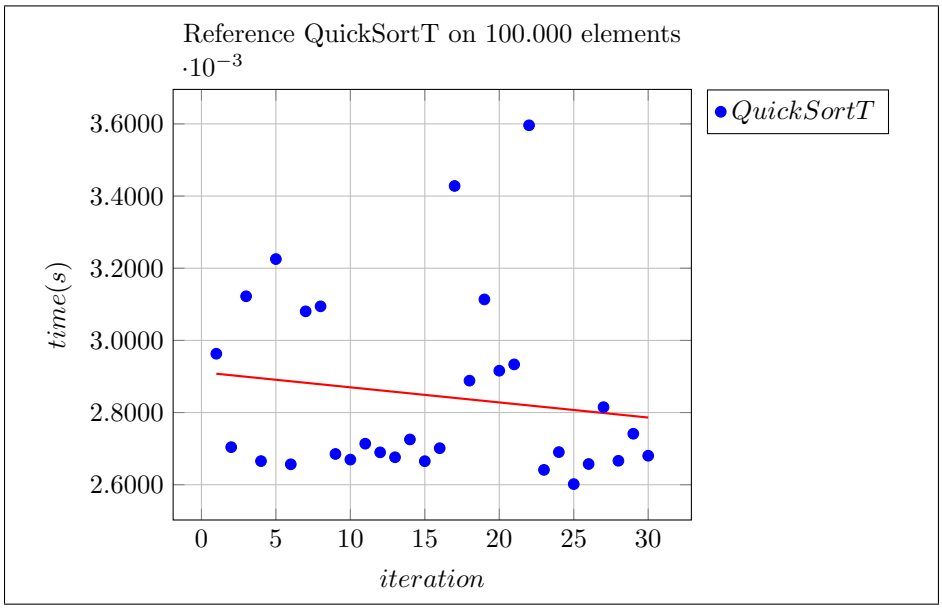

Figure E74: QuickSortT scatter plot on 100.000 elements.

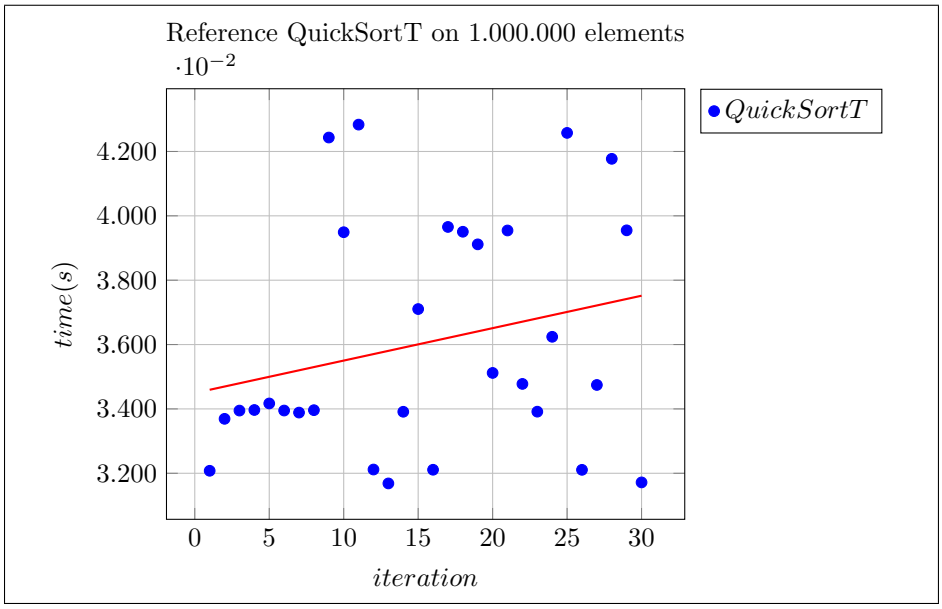

Figure E75: QuickSortT scatter plot on 1.000.000 elements.

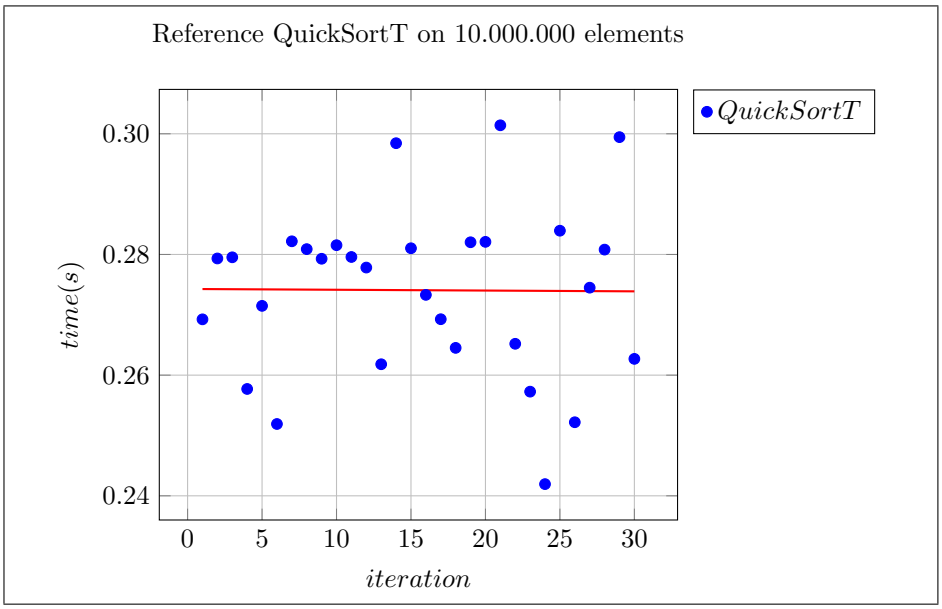

Figure E76: QuickSortT scatter plot on 10.000.000 elements.

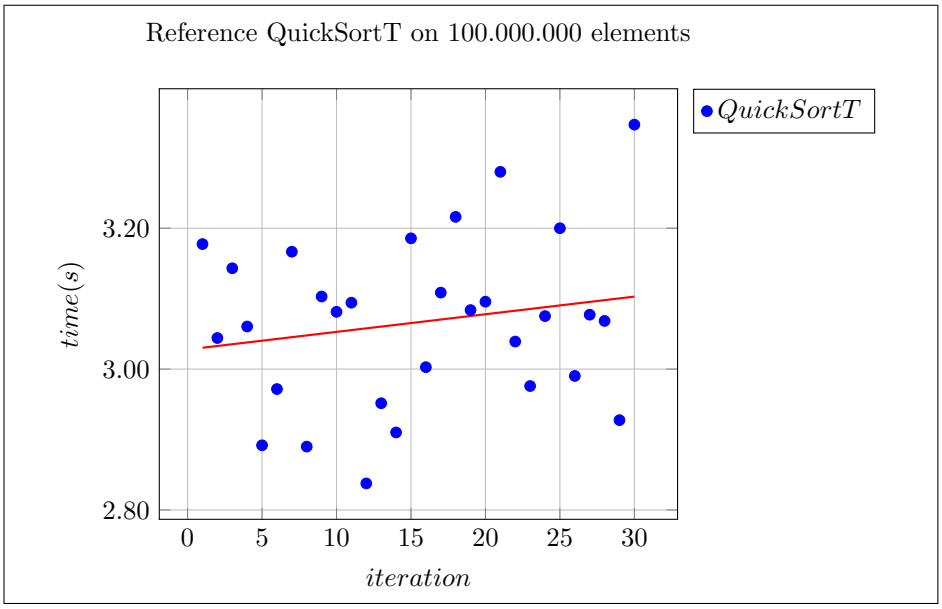

Figure E77: QuickSortT scatter plot on 100.000.000 elements.

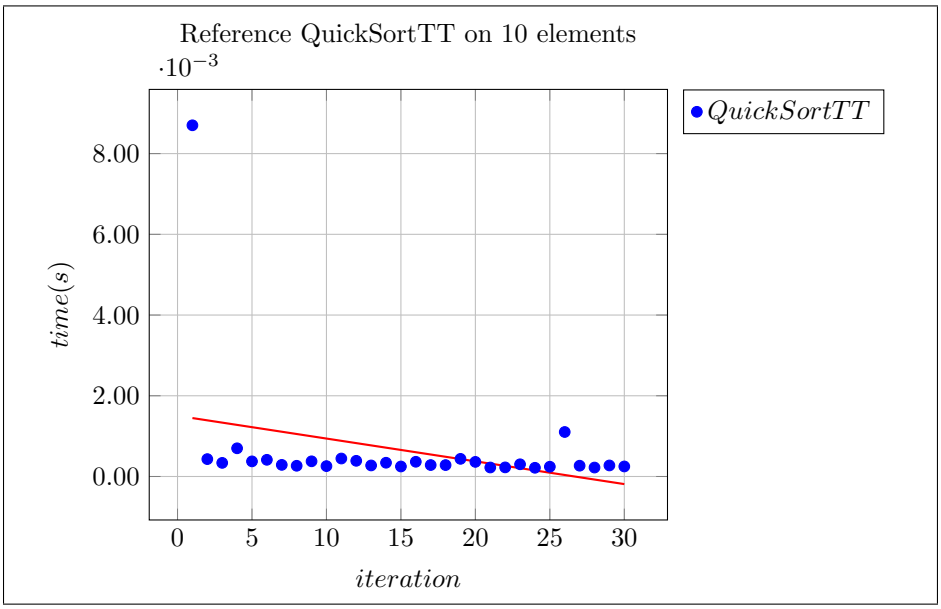

Figure E78: QuickSortTT scatter plot on 10 elements.

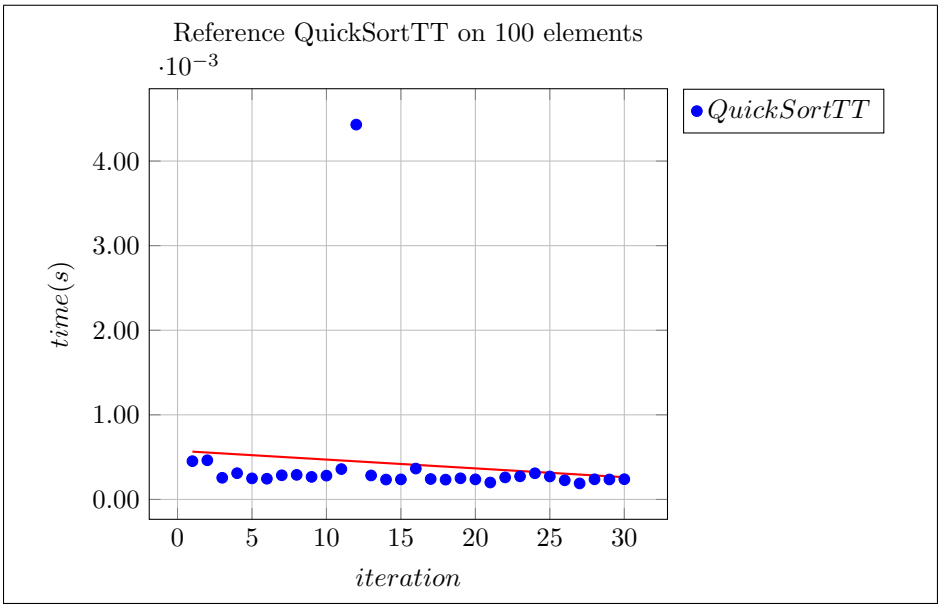

Figure E79: QuickSortTT scatter plot on 100 elements.

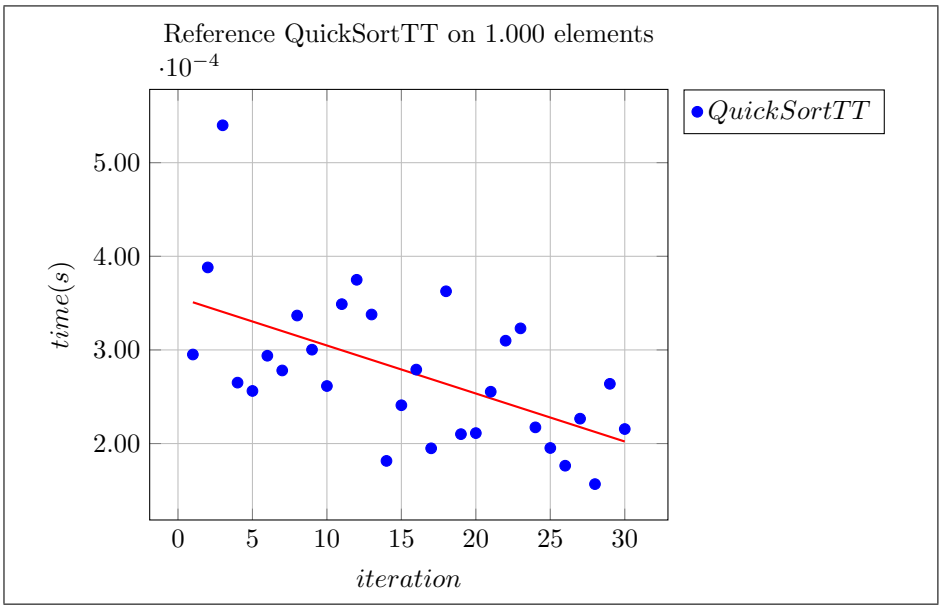

Figure E80: QuickSortTT scatter plot on 1.000 elements.

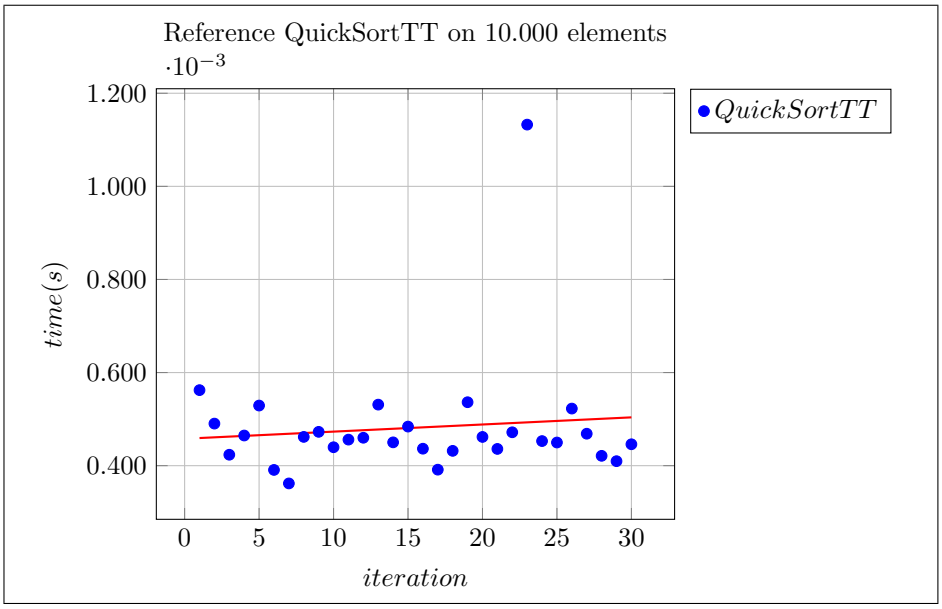

Figure E81: QuickSortTT scatter plot on 10.000 elements.

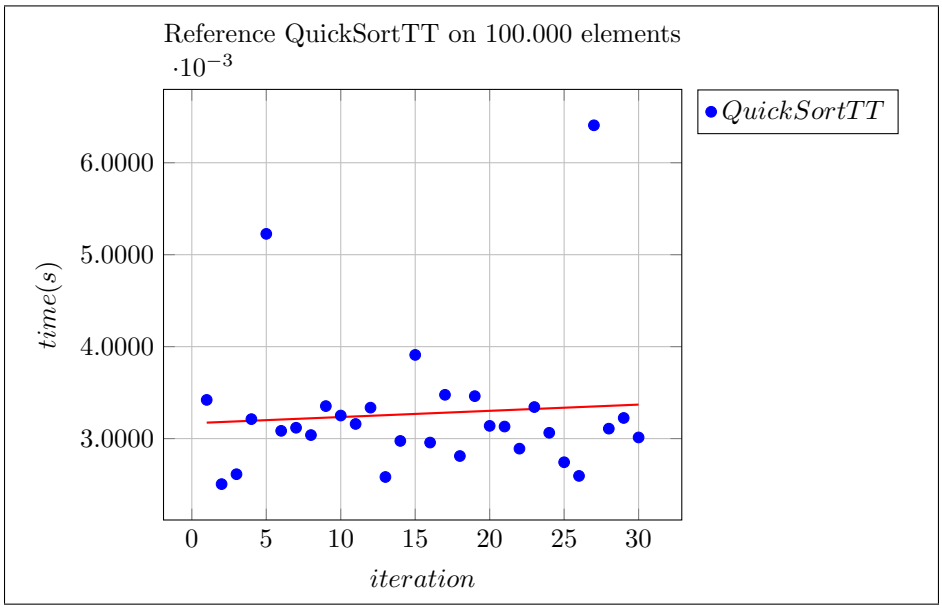

Figure E82: QuickSortTT scatter plot on 100.000 elements.

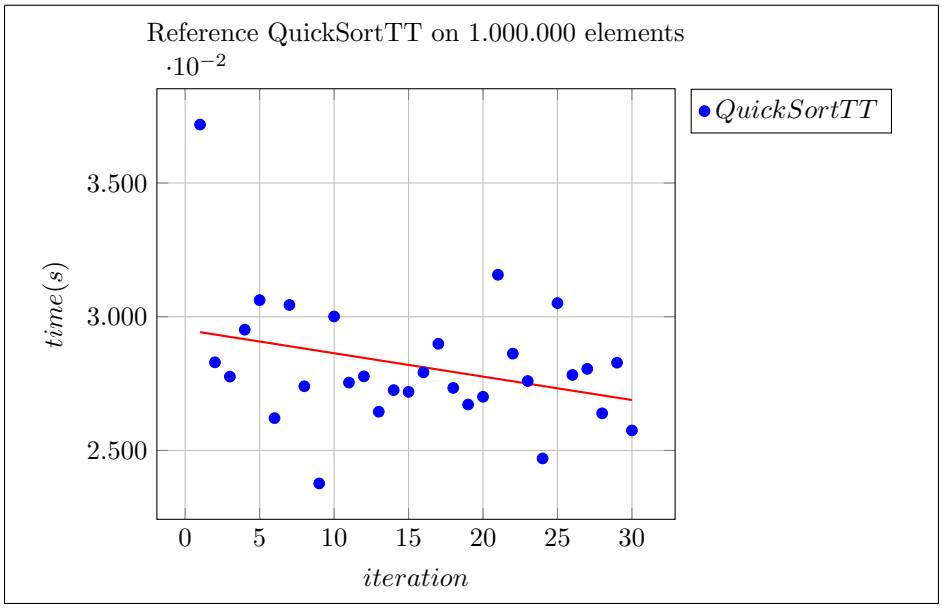

Figure E83: QuickSortTT scatter plot on 1.000.000 elements.

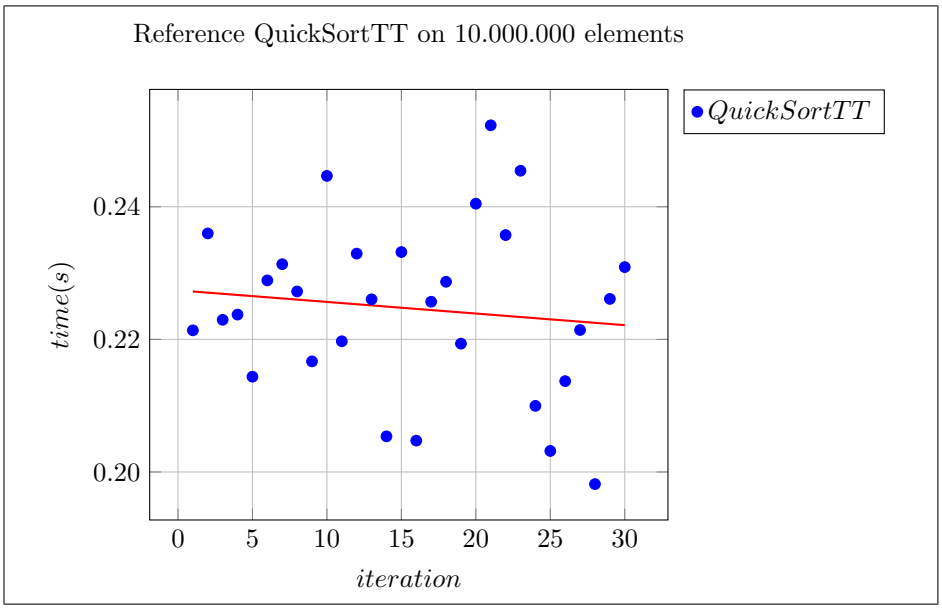

Figure E84: QuickSortTT scatter plot on 10.000.000 elements.

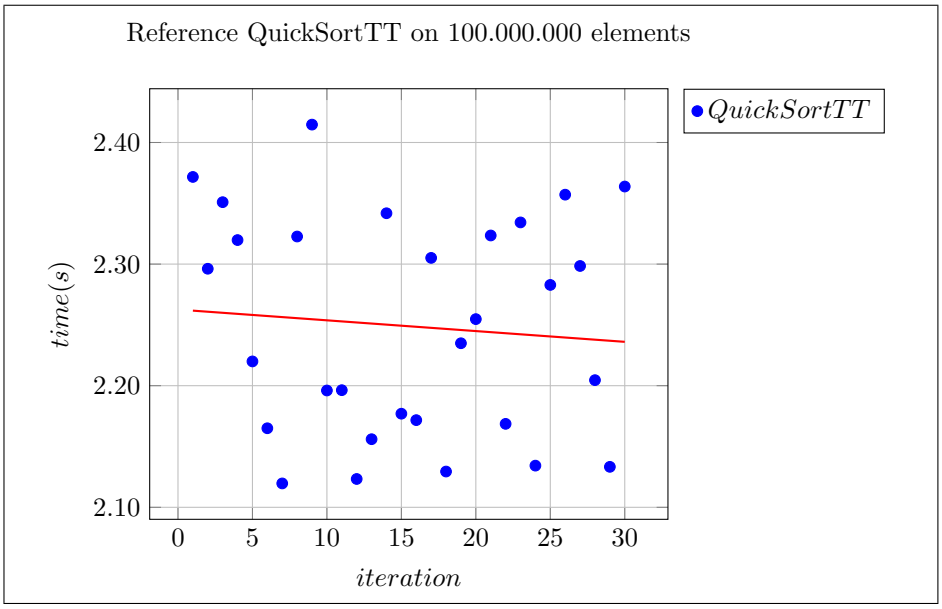

Figure E85: QuickSortTT scatter plot on 100.000.000 elements.

E.5 QuickSortT Virtualized

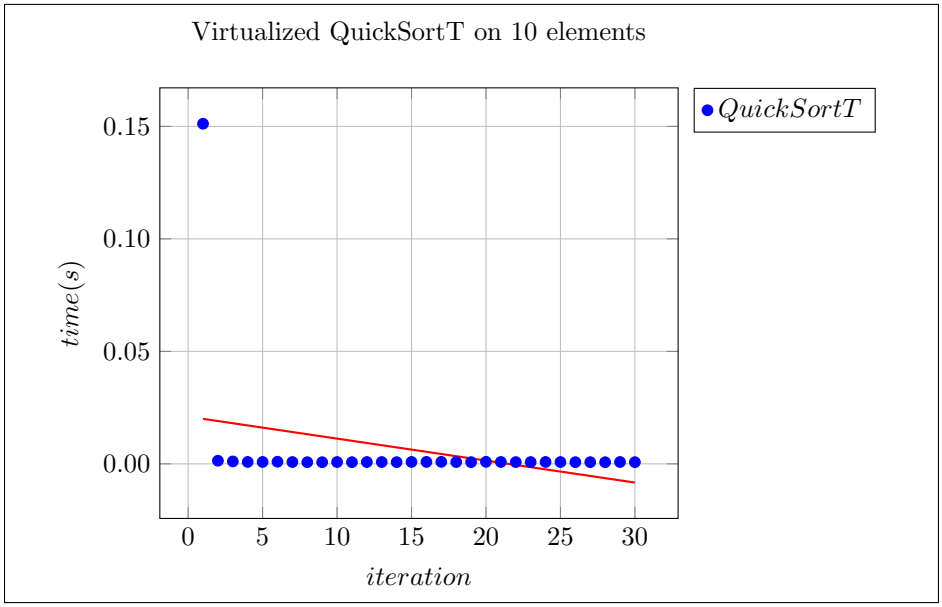

Figure E86: Virtualized QuickSortT scatter plot on 10 elements.

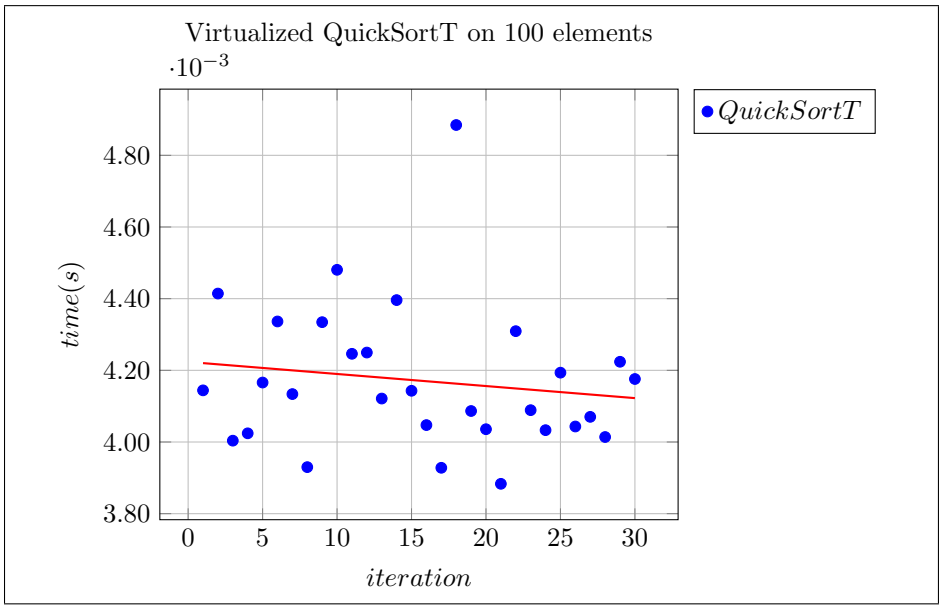

Figure E87: Virtualized QuickSortT scatter plot on 100 elements.

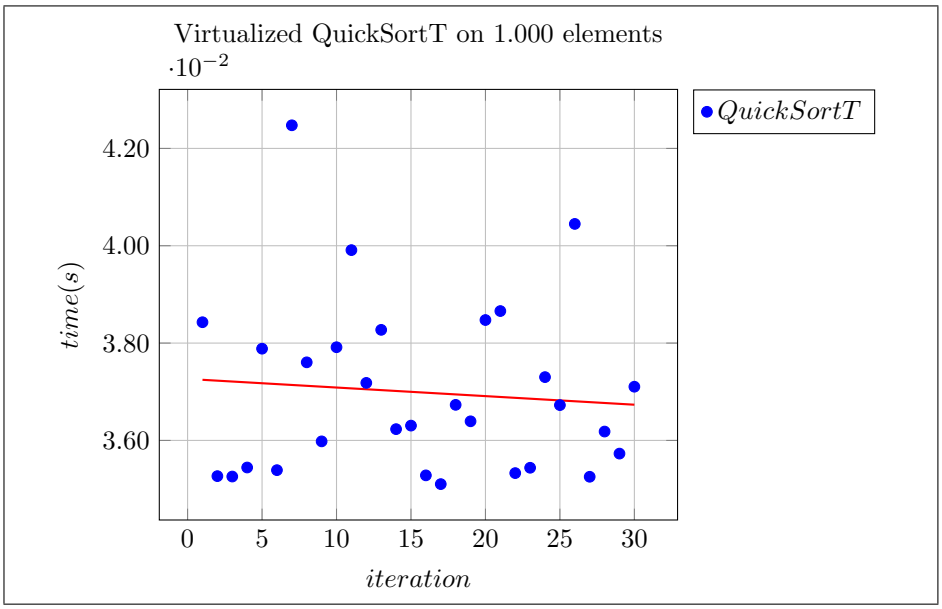

Figure E88: Virtualized QuickSortT scatter plot on 1.000 elements.

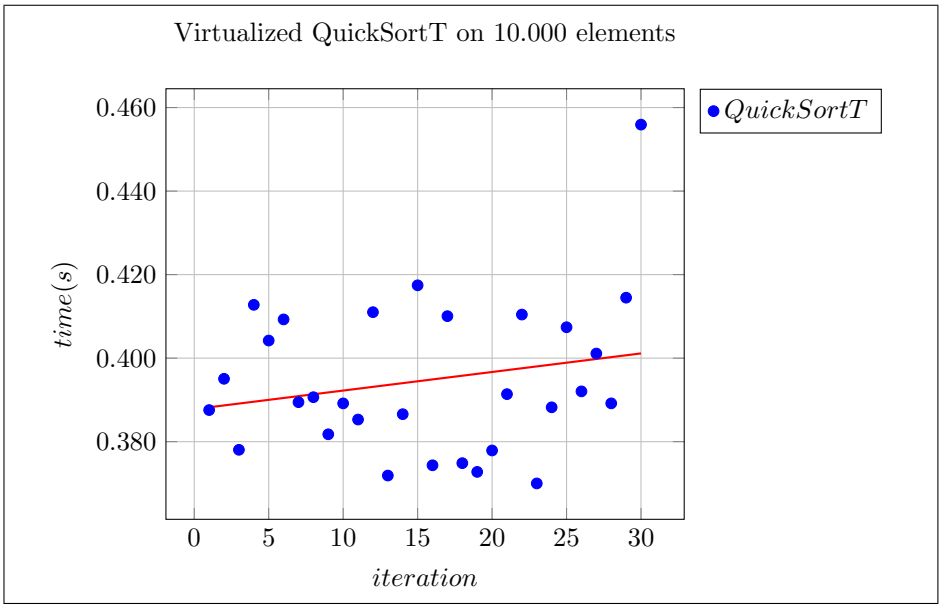

Figure E89: Virtualized QuickSortT scatter plot on 10.000 elements.

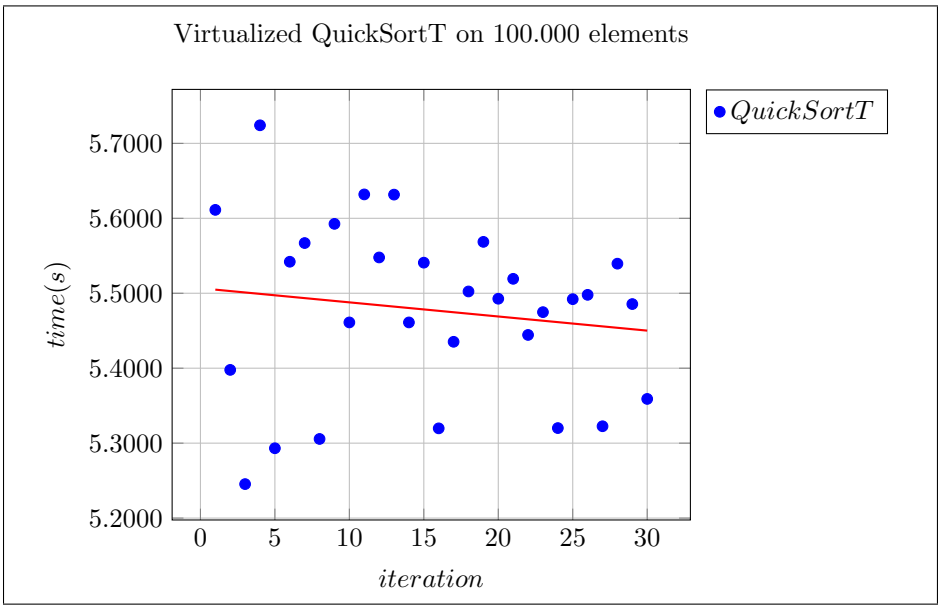

Figure E90: Virtualized QuickSortT scatter plot on 100.000 elements.

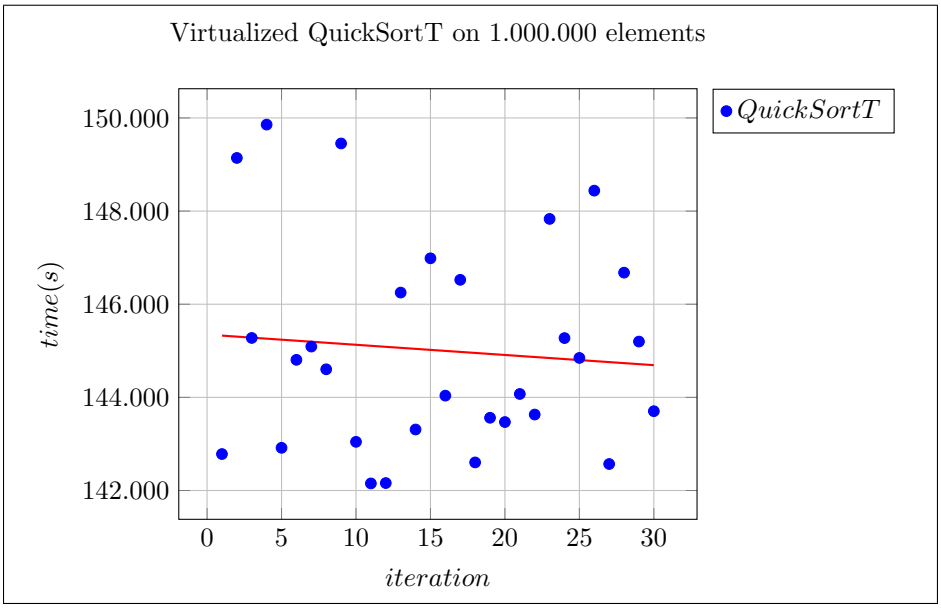

Figure E91: Virtualized QuickSortT scatter plot on 1.000.000 elements.

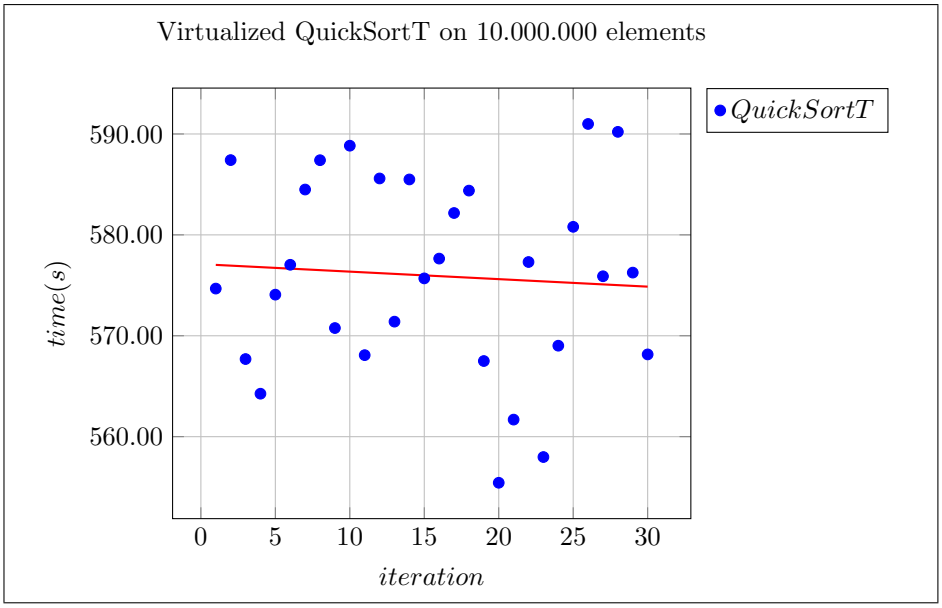

Figure E92: Virtualized QuickSortT scatter plot on 10.000.000 elements.

E.6 QuickSortTT Virtualized

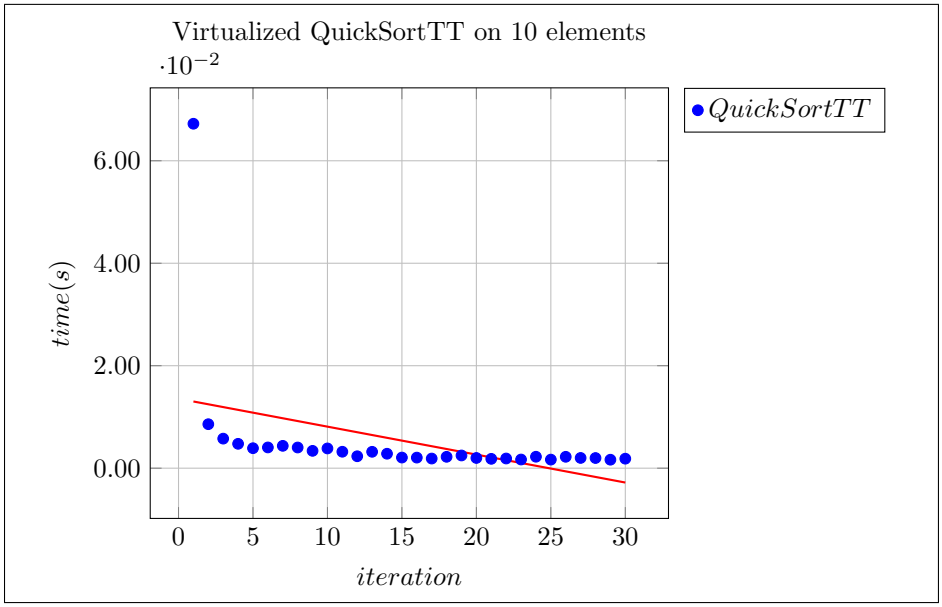

Figure E93: Virtualized QuickSortTT scatter plot on 10 elements.

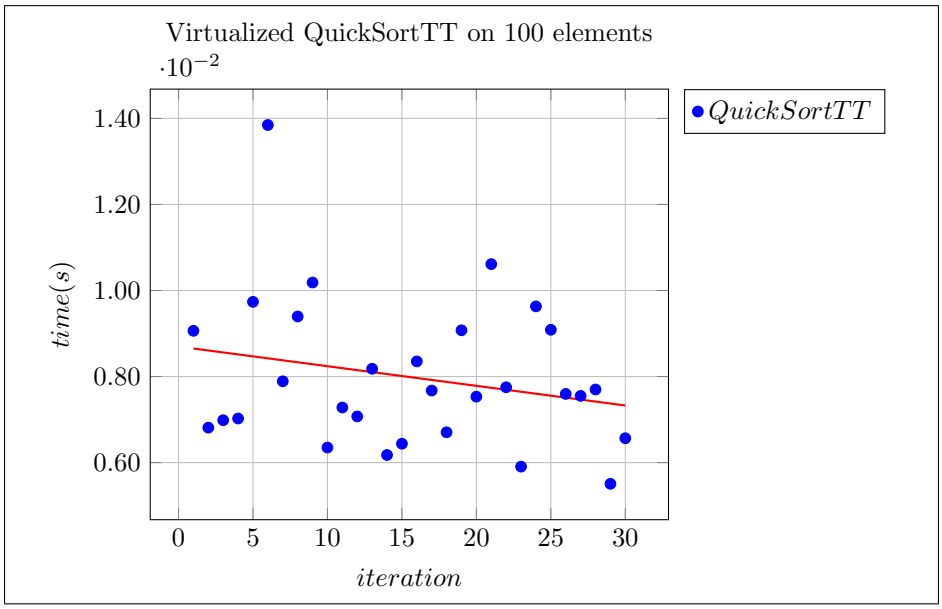

Figure E94: Virtualized QuickSortTT scatter plot on 100 elements.

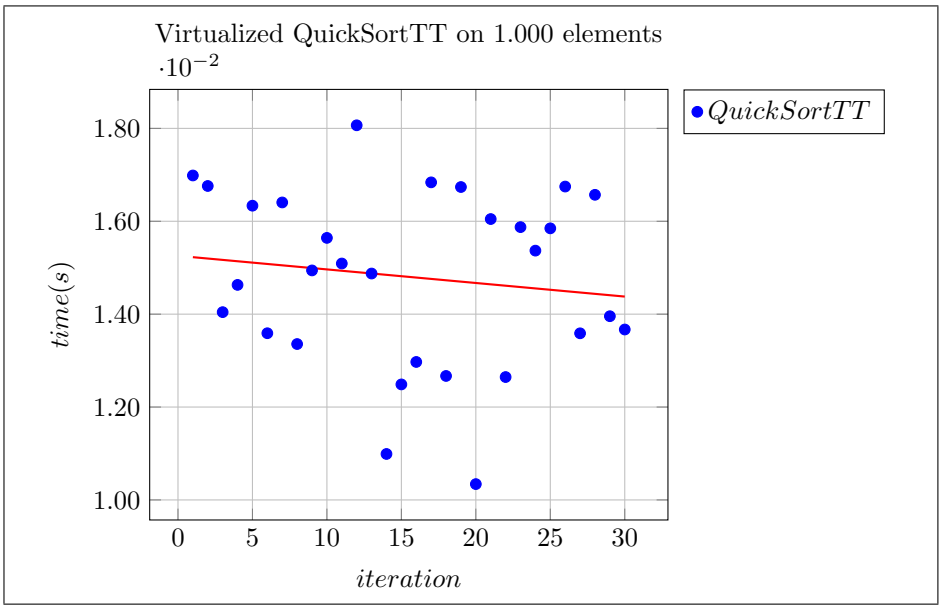

Figure E95: Virtualized QuickSortTT scatter plot on 1.000 elements.

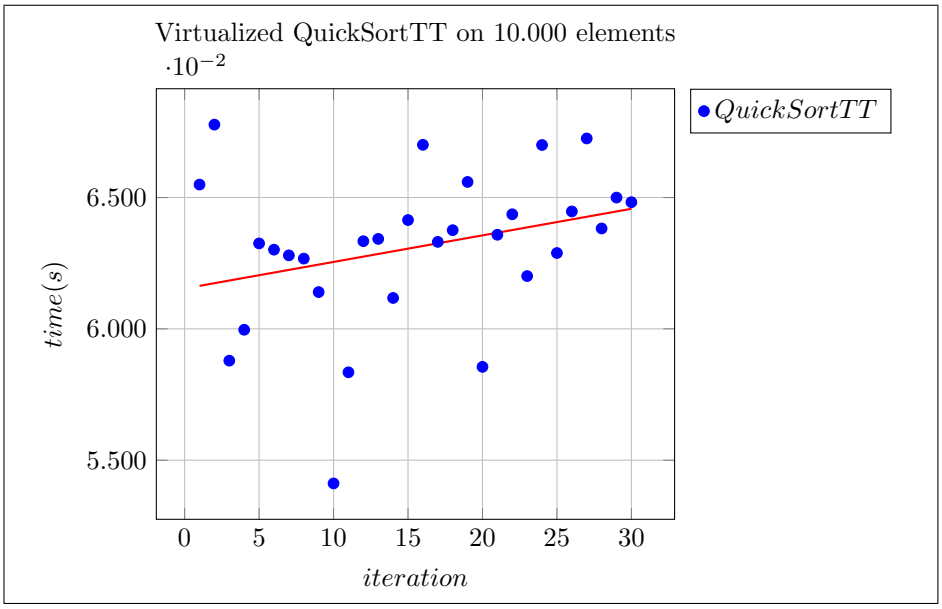

Figure E96: Virtualized QuickSortTT scatter plot on 10.000 elements.

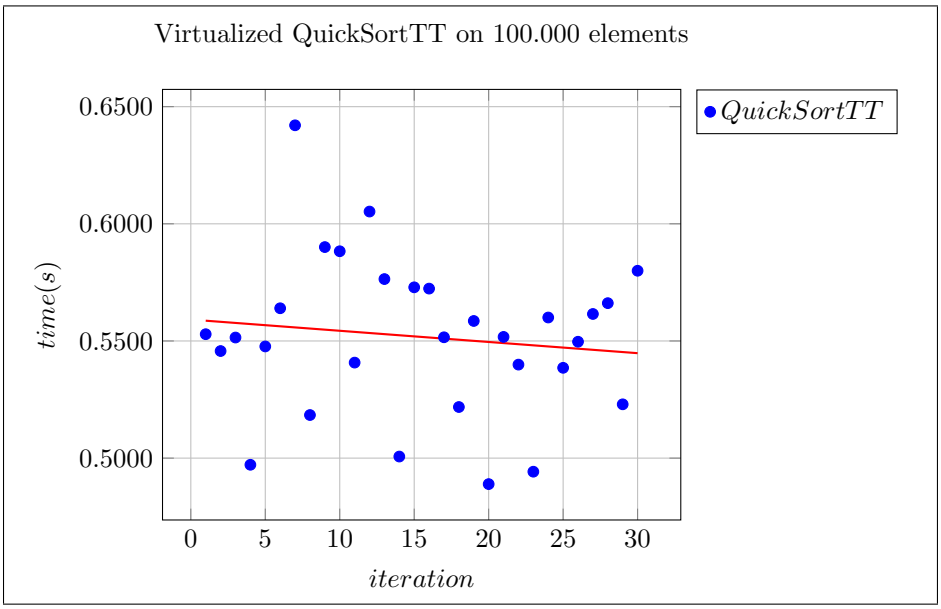

Figure E97: Virtualized QuickSortTT scatter plot on 100.000 elements.

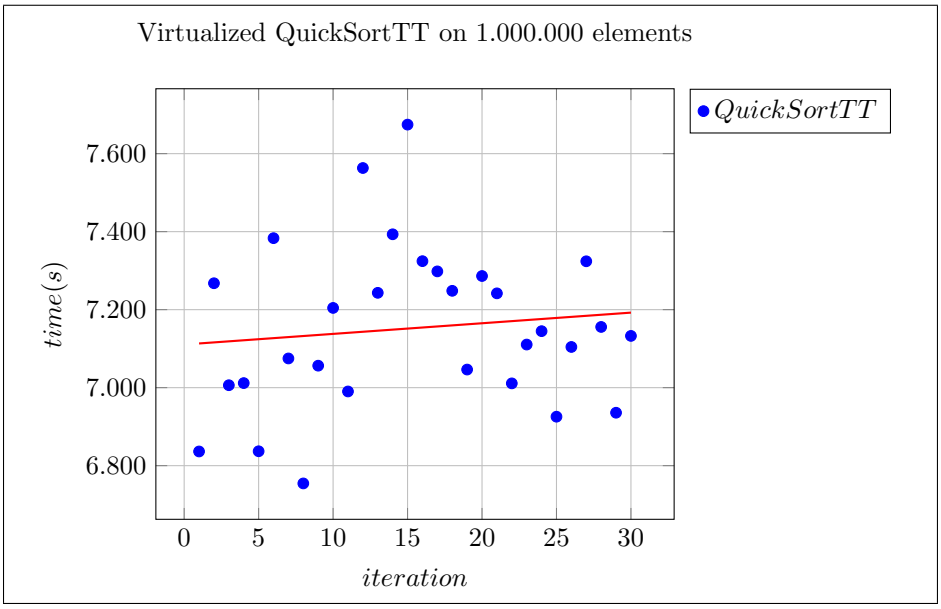

Figure E98: Virtualized QuickSortTT scatter plot on 1.000.000 elements.

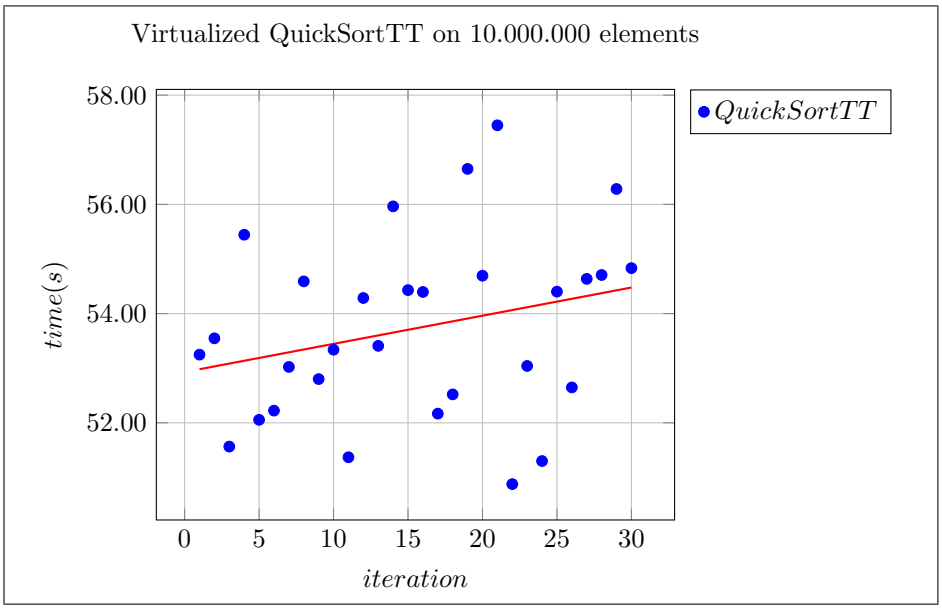

Figure E99: Virtualized QuickSortTT scatter plot on 10.000.000 elements.

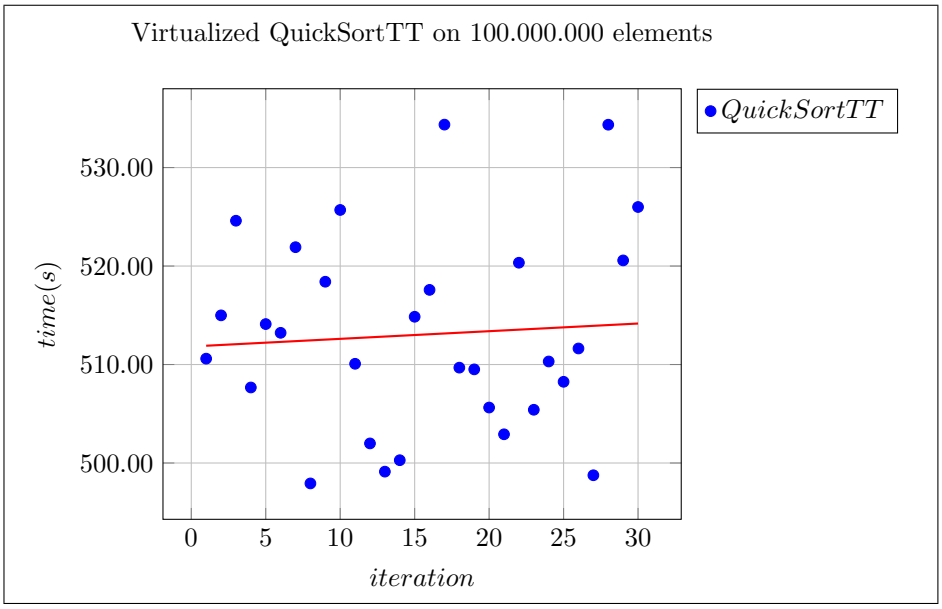

Figure E100: Virtualized QuickSortTT scatter plot on 100.000.000 elements.

Appendix F

Bundle Runtimes

F.1 Sorting Algorithm Runtimes

	Reference		Virtualized		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$6{\cdot}10^{-5}$	$5.8 \cdot 10^{-5}$	$3.95 \cdot 10^{-4}$	$2.8 \cdot 10^{-4}$	6.6
$1 \cdot 10^2$	$1.21 \cdot 10^{-4}$	$9.2 \cdot 10^{-5}$	$2.61 \cdot 10^{-2}$	$8{\cdot}10^{-5}$	216.3
$1 \cdot 10^3$	$9.49 \cdot 10^{-4}$	$6{\cdot}10^{-6}$	2.83	$1.1 \cdot 10^{-3}$	2,976.3
$1 \cdot 10^4$	0.11	$9.2 \cdot 10^{-5}$	285.65	$8.7 \cdot 10^{-2}$	2,494.4

Table F1: BubbleSort measurements in OSGi.

	Reference		Virtualized		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$5.8 \cdot 10^{-5}$	$2.3 \cdot 10^{-5}$	$2.72 \cdot 10^{-4}$	$1.8 \cdot 10^{-4}$	4.7
$1 \cdot 10^2$	$5.5 \cdot 10^{-5}$	$8{\cdot}10^{-6}$	$9.65 \cdot 10^{-4}$	$1.3 \cdot 10^{-4}$	17.6
$1 \cdot 10^3$	$7{\cdot}10^{-5}$	$3.5 \cdot 10^{-5}$	$8.34 \cdot 10^{-3}$	$4.1 \cdot 10^{-5}$	118.3
$1 \cdot 10^4$	$1.28 \cdot 10^{-4}$	$7{\cdot}10^{-6}$	$8.22 \cdot 10^{-2}$	$2.6 \cdot 10^{-4}$	643.1
$1 \cdot 10^5$	$9.87 \cdot 10^{-4}$	$4.6 \cdot 10^{-5}$	0.82	$5.1 \cdot 10^{-4}$	831.8
$1 \cdot 10^6$	$2.92 \cdot 10^{-3}$	$1.1 \cdot 10^{-5}$	8.19	$4.7 \cdot 10^{-3}$	2,800.8
$1 \cdot 10^7$	0.15	$2.2 \cdot 10^{-4}$	82.78	$7.1 \cdot 10^{-2}$	545.2
$1 \cdot 10^8$	1.88	$1{\cdot}10^{-2}$	828.87	$4.6 \cdot 10^{-1}$	439.9

Table F2: BucketSort measurements in OSGi.

	Reference		Virtualized		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$7{\cdot}10^{-5}$	$7.3 \cdot 10^{-5}$	$2.85 \cdot 10^{-4}$	$1.8 \cdot 10^{-4}$	4.1
$1 \cdot 10^2$	$5.8 \cdot 10^{-5}$	$1{\cdot}10^{-5}$	$1.93 \cdot 10^{-3}$	$3.7 \cdot 10^{-4}$	33.5
$1 \cdot 10^3$	$1.12 \cdot 10^{-4}$	$5{\cdot}10^{-6}$	$2.25 \cdot 10^{-2}$	$1.7 \cdot 10^{-4}$	200.6
$1 \cdot 10^4$	$7.86 \cdot 10^{-4}$	$6{\cdot}10^{-6}$	0.27	$1.8 \cdot 10^{-4}$	347.7
$1 \cdot 10^5$	$8.75 \cdot 10^{-3}$	$3.5 \cdot 10^{-5}$	3.32	$7.1 \cdot 10^{-4}$	379.1
$1 \cdot 10^6$	$7.47 \cdot 10^{-2}$	$7.3 \cdot 10^{-5}$	35.8	$1.4 \cdot 10^{-2}$	479
$1 \cdot 10^7$	1.12	$2.1 \cdot 10^{-4}$	428.58	$1.7 \cdot 10^{-1}$	382.1
$1 \cdot 10^8$	12.79	$4.4 \cdot 10^{-3}$	4,803.44	$8.7 \cdot 10^{-1}$	375.5

Table F3: QuickSort measurements in OSGi.

	Reference		Virtualized		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$2.13 \cdot 10^{-4}$	$3.9 \cdot 10^{-4}$	$4.41 \cdot 10^{-4}$	$5.2 \cdot 10^{-4}$	2.1
$1 \cdot 10^2$	$2.28 \cdot 10^{-4}$	$3{\cdot}10^{-5}$	$1.77 \cdot 10^{-3}$	$1.7 \cdot 10^{-4}$	7.8
$1 \cdot 10^3$	$1.86 \cdot 10^{-4}$	$1.9 \cdot 10^{-5}$	$1.74 \cdot 10^{-2}$	$6.7 \cdot 10^{-4}$	93.9
$1 \cdot 10^4$	$3.65 \cdot 10^{-4}$	$2.8 \cdot 10^{-5}$	0.2	$5.7 \cdot 10^{-3}$	535.5
$1 \cdot 10^5$	$2.7 \cdot 10^{-3}$	$2.2 \cdot 10^{-4}$	2.65	$3.4 \cdot 10^{-2}$	982
$1 \cdot 10^6$	$3.67 \cdot 10^{-2}$	$3.5 \cdot 10^{-3}$	71.59	$8.7 \cdot 10^{-1}$	1,948.5
$1 \cdot 10^7$	0.27	$1.4 \cdot 10^{-2}$	293.67	3	1,089.3
$1 \cdot 10^8$	3.03	$1{\cdot}10^{-1}$	3,531.63	12.4	1,164.8

Table F4: QuickSortT measurements in OSGi.

	Reference		Virtualized		
elements	(s)	σ	(s)	σ	*factor*
$1 \cdot 10^1$	$2.32 \cdot 10^{-4}$	$4{\cdot}10^{-4}$	$5.78 \cdot 10^{-4}$	$4.2 \cdot 10^{-4}$	2.5
$1 \cdot 10^2$	$1.87 \cdot 10^{-4}$	$8{\cdot}10^{-5}$	$1.03 \cdot 10^{-3}$	$1.6 \cdot 10^{-4}$	5.5
$1 \cdot 10^3$	$2.32 \cdot 10^{-4}$	$7{\cdot}10^{-5}$	$6.68 \cdot 10^{-3}$	$5{\cdot}10^{-4}$	28.8
$1 \cdot 10^4$	$5{\cdot}10^{-4}$	$1{\cdot}10^{-4}$	$6.12 \cdot 10^{-2}$	$7.7 \cdot 10^{-4}$	122.3
$1 \cdot 10^5$	$3.24 \cdot 10^{-3}$	$8.3 \cdot 10^{-4}$	0.6	$2{\cdot}10^{-3}$	186.6
$1 \cdot 10^6$	$2.76 \cdot 10^{-2}$	$3.3 \cdot 10^{-3}$	8.89	$1.9 \cdot 10^{-2}$	322.4
$1 \cdot 10^7$	0.22	$1.3 \cdot 10^{-2}$	60.4	$7.1 \cdot 10^{-2}$	272.6
$1 \cdot 10^8$	2.33	$3{\cdot}10^{-1}$	606.85	4.3	260.6

Table F5: QuickSortTT measurements in OSGi.

Appendix G

Sorting Program Metrics

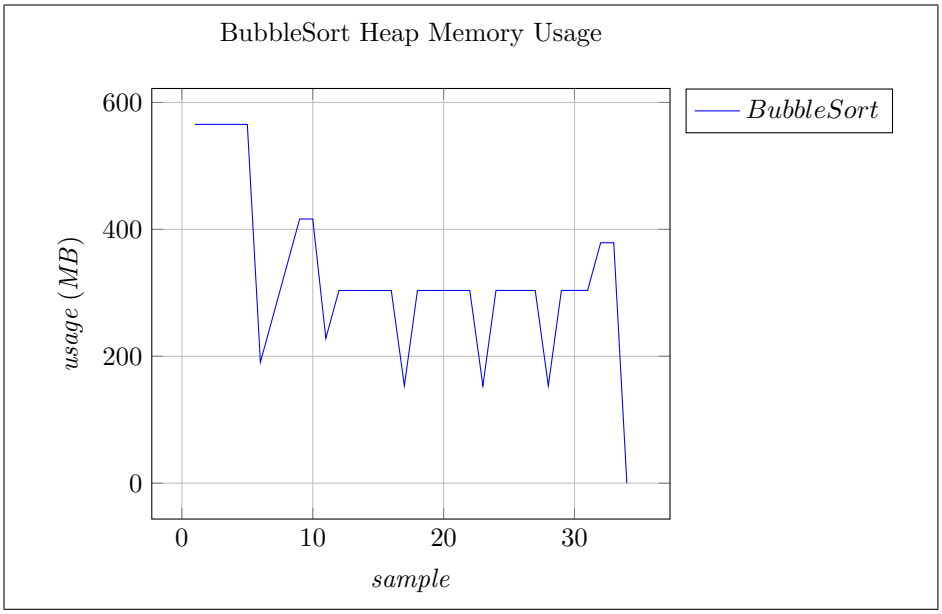

Figure G1: BubbleSort heap metrics - 100.000 elements.

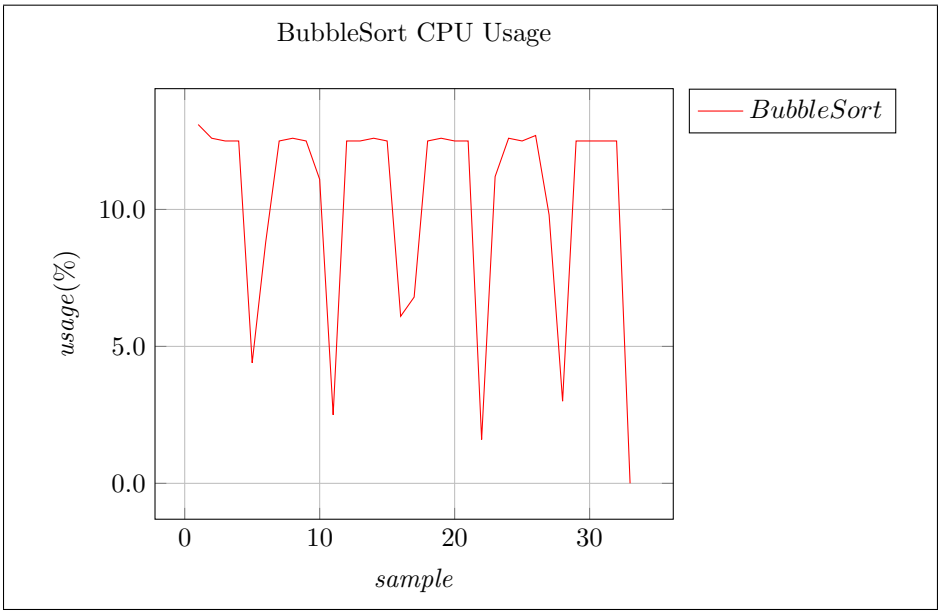

Figure G2: BubbleSort CPU Metrics - 100.000 elements.

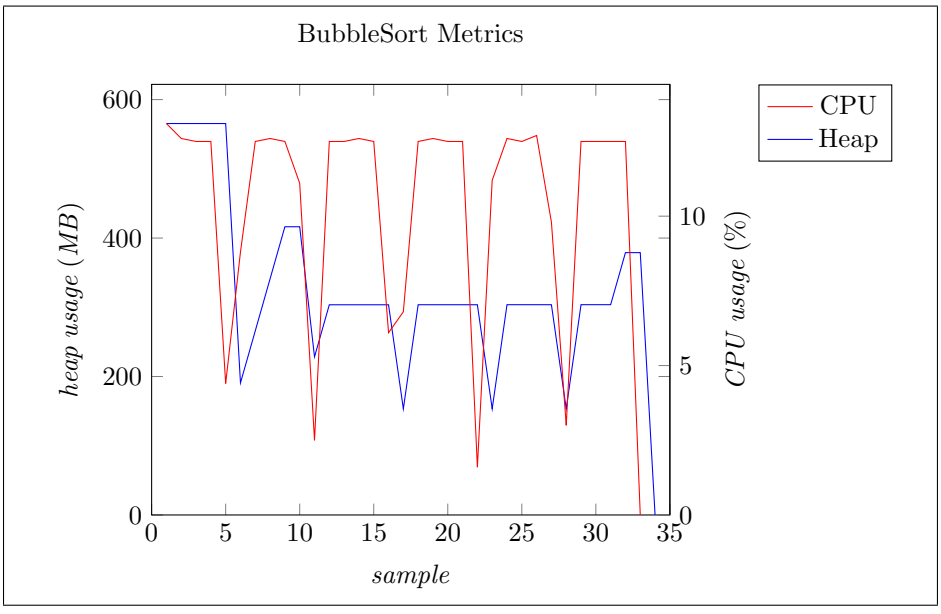

Figure G3: BubbleSort metrics combined - 100.000 elements.

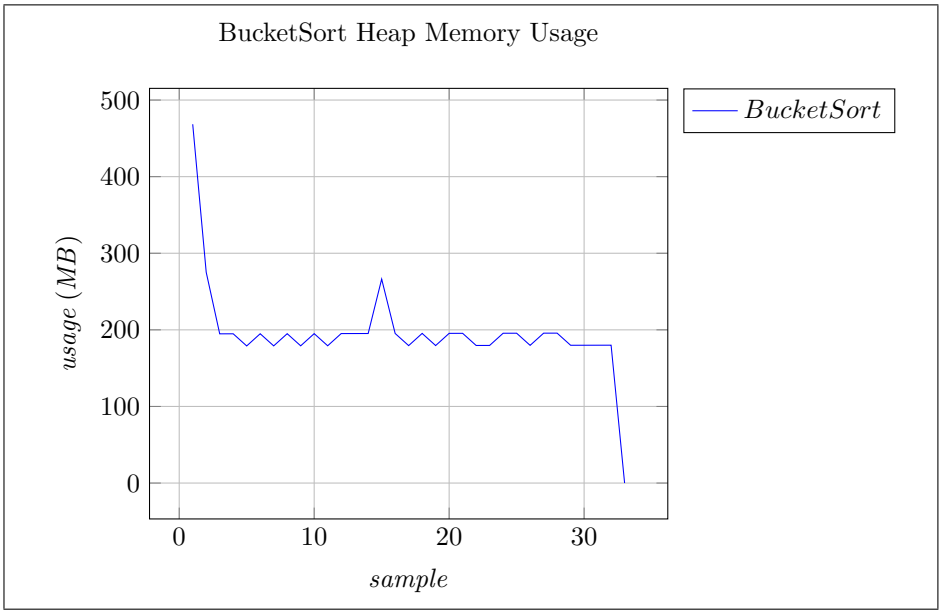

Figure G4: BucketSort heap metrics - 10.000.000 elements.

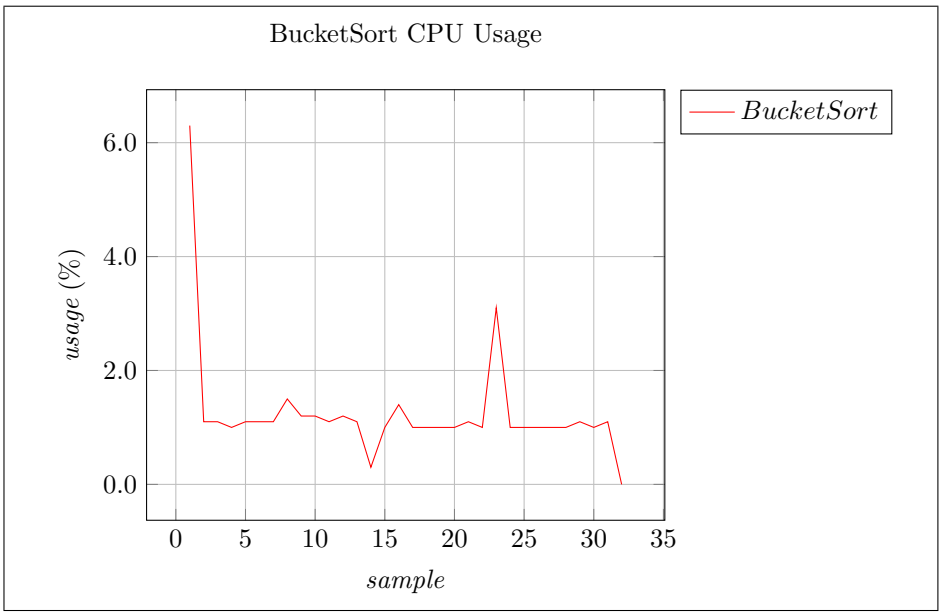

Figure G5: BucketSort CPU metrics - 10.000.000 elements.

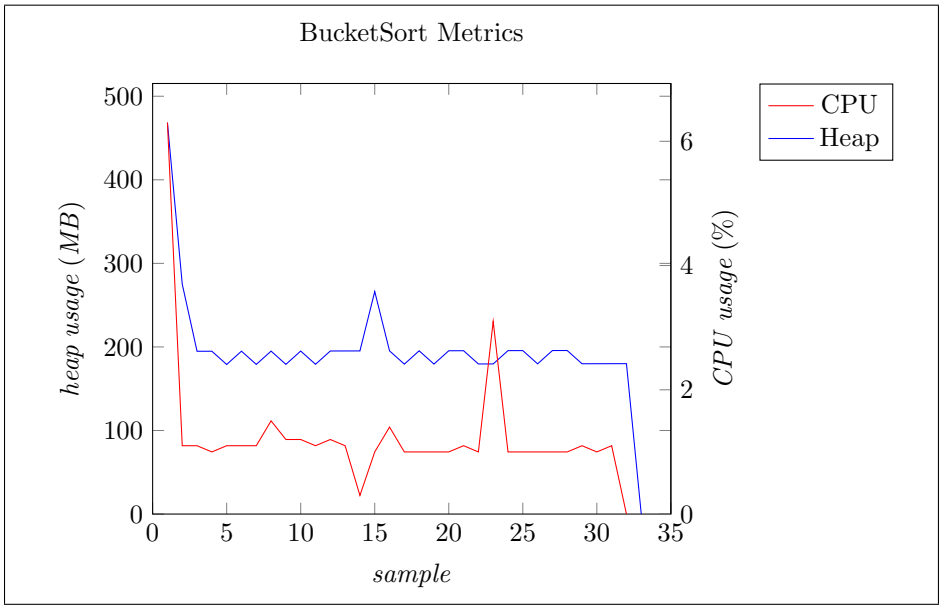

Figure G6: BucketSort metrics combined - 10.000.000 elements.

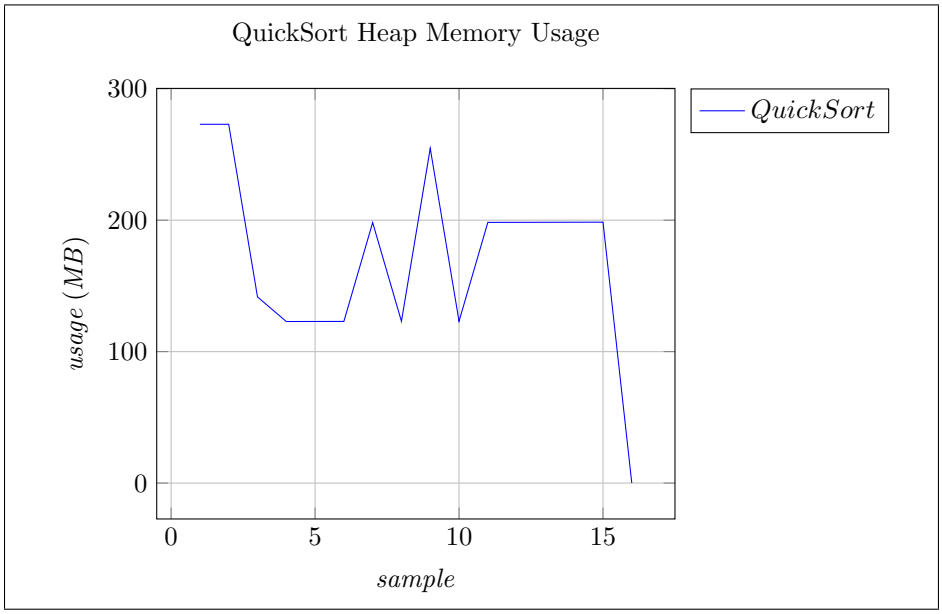

Figure G7: QuickSort heap metrics - 1.000.000 elements.

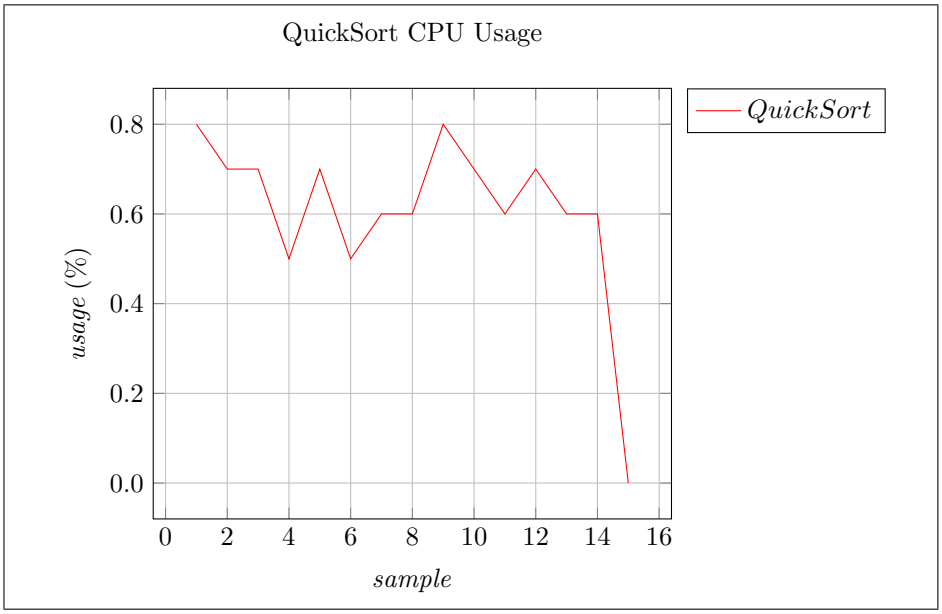

Figure G8: QuickSort CPU metrics - 1.000.000 elements.

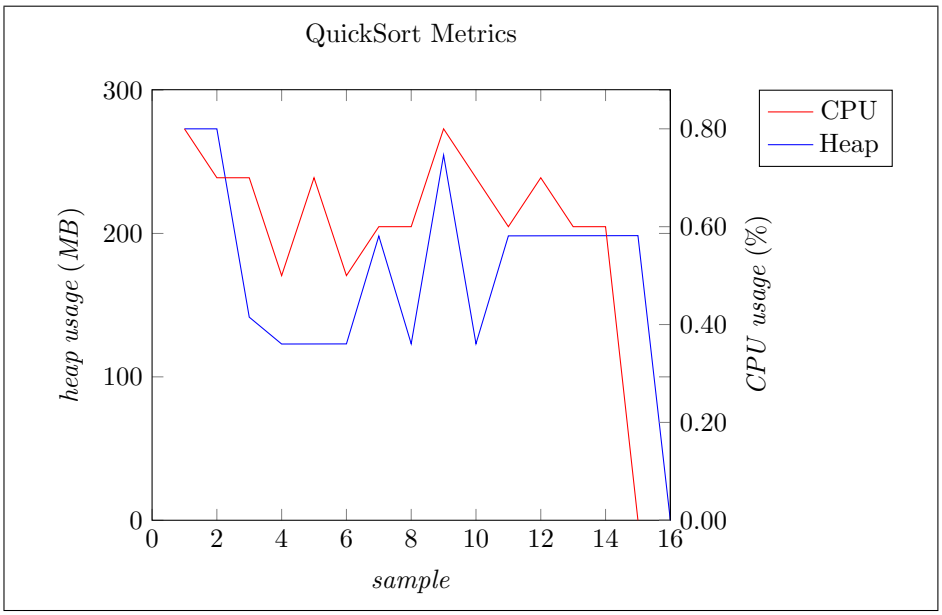

Figure G9: QuickSort combined metrics - 1.000.000 elements.

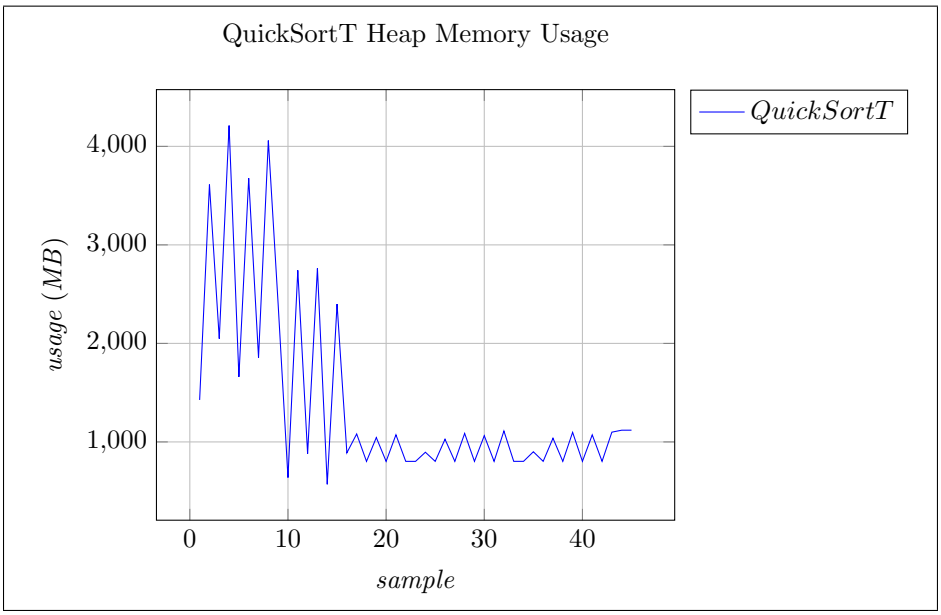

Figure G10: QuickSortT heap metrics - 100.000.000 elements.

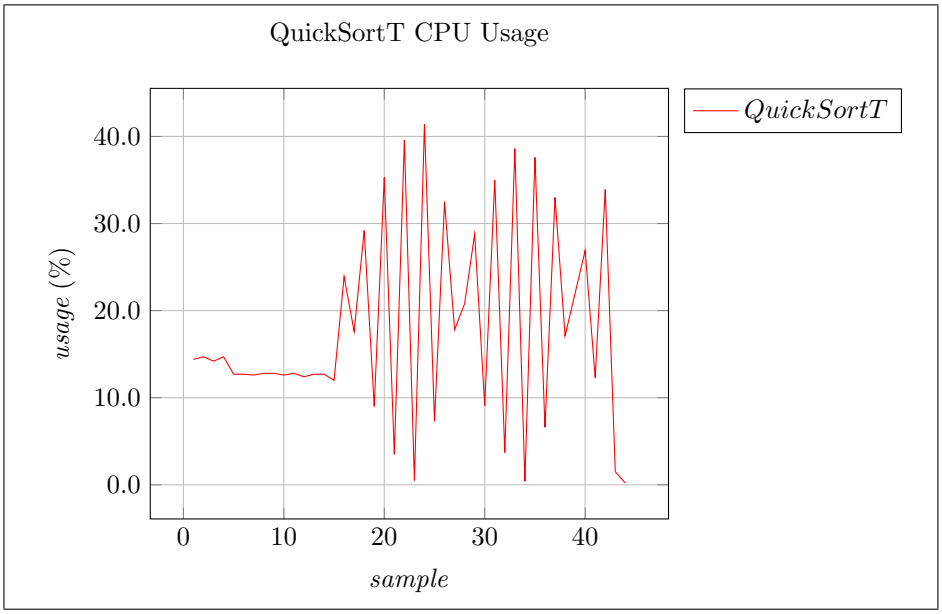

Figure G11: QuickSortT CPU metrics - 100.000.000 elements.

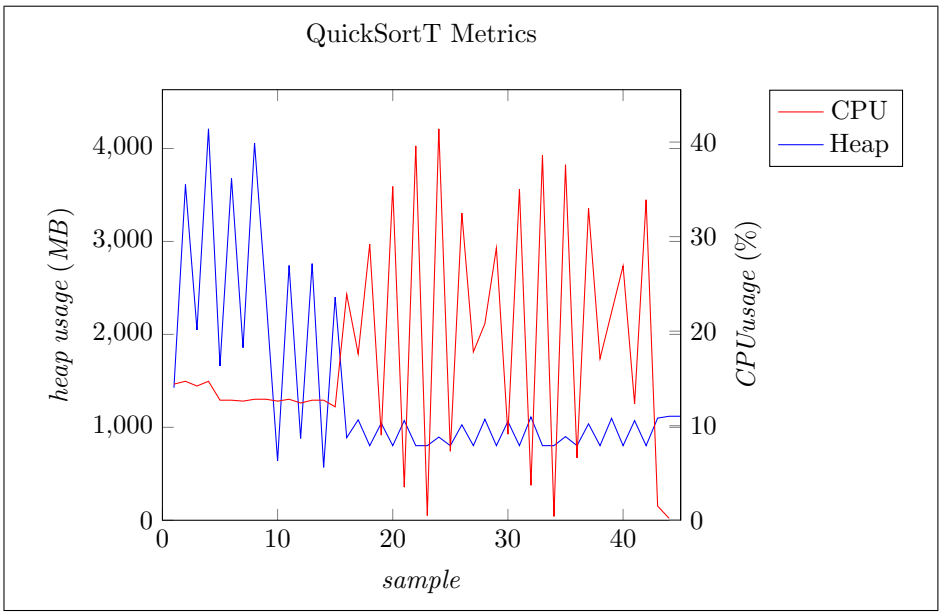

Figure G12: QuickSortT combined metrics - 100.000.000 elements.

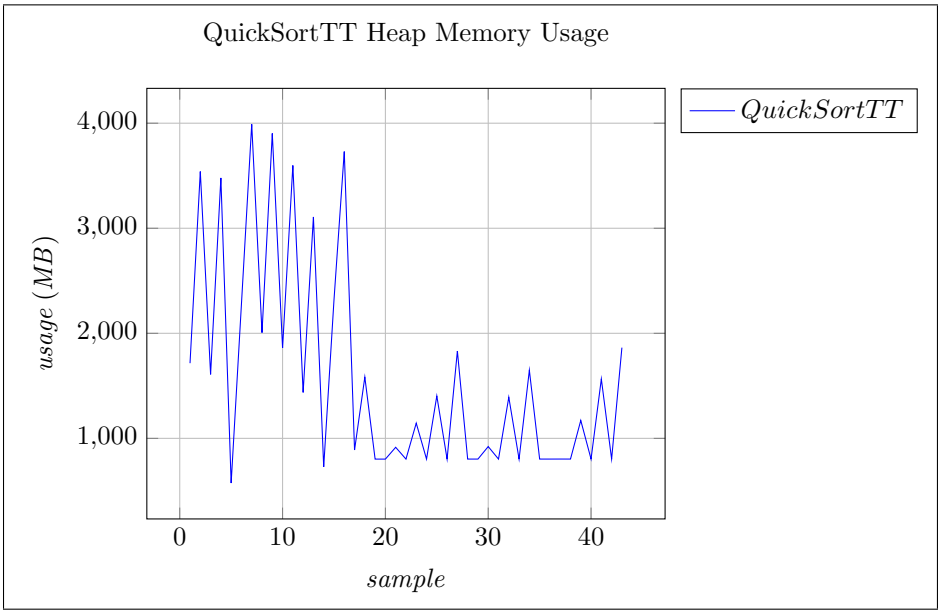

Figure G13: QuickSortTT heap metrics - 100.000.000 elements.

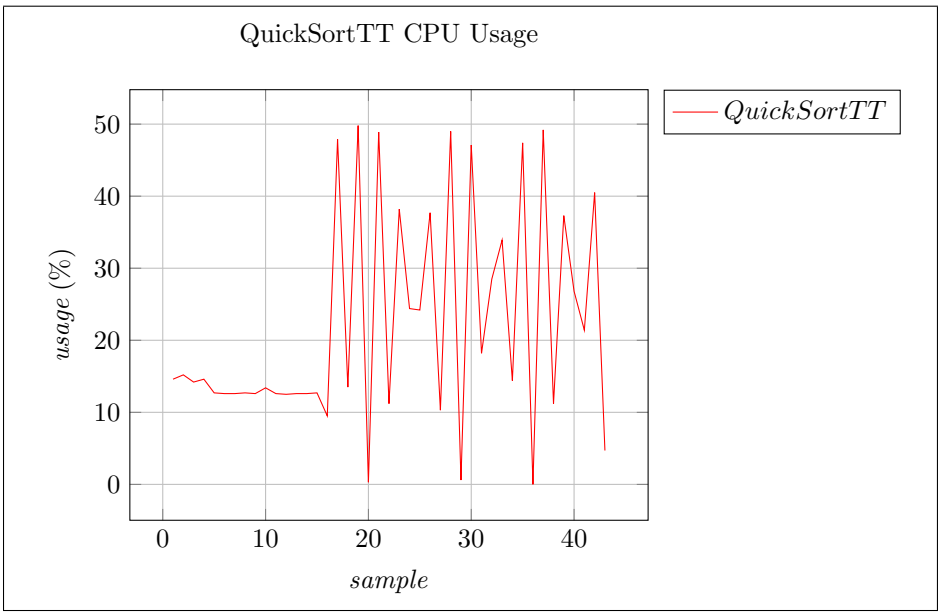

Figure G14: QuickSortTT CPU metrics - 100.000.000 elements.

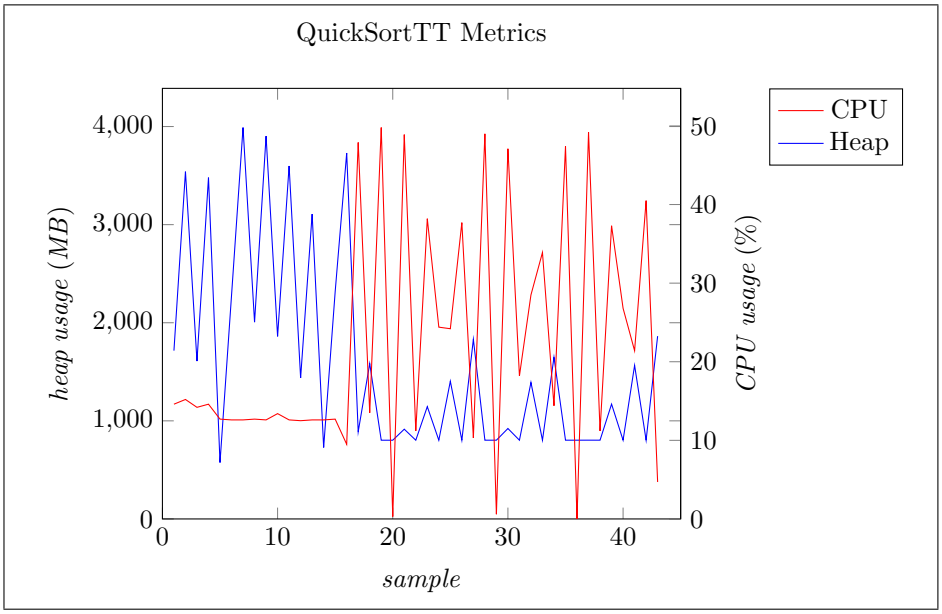

Figure G15: QuickSortTT combined metrics. - 100.000.000 elements

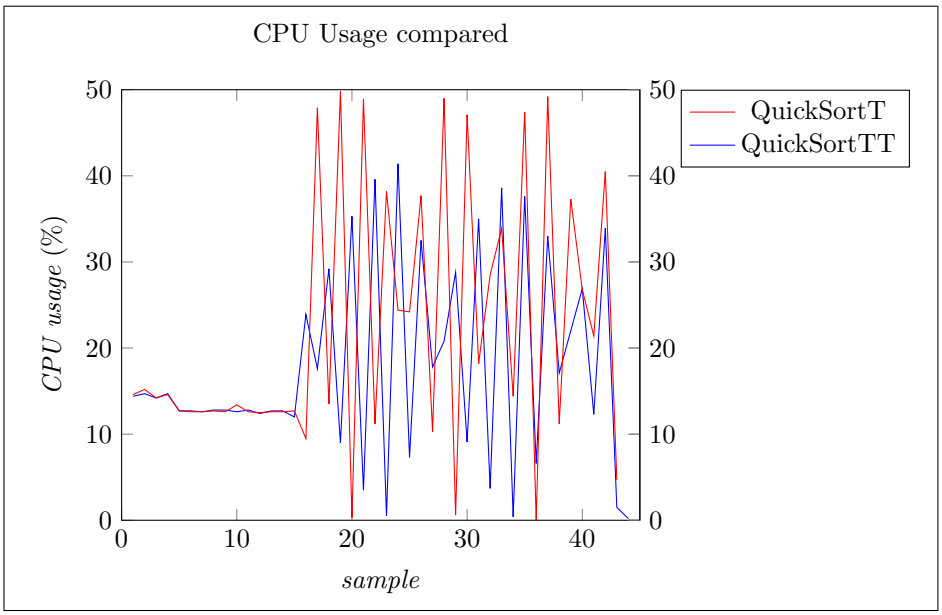

Figure G16: QuickSortT vs QuickSortTT CPU usage - 100.000.000 elements.

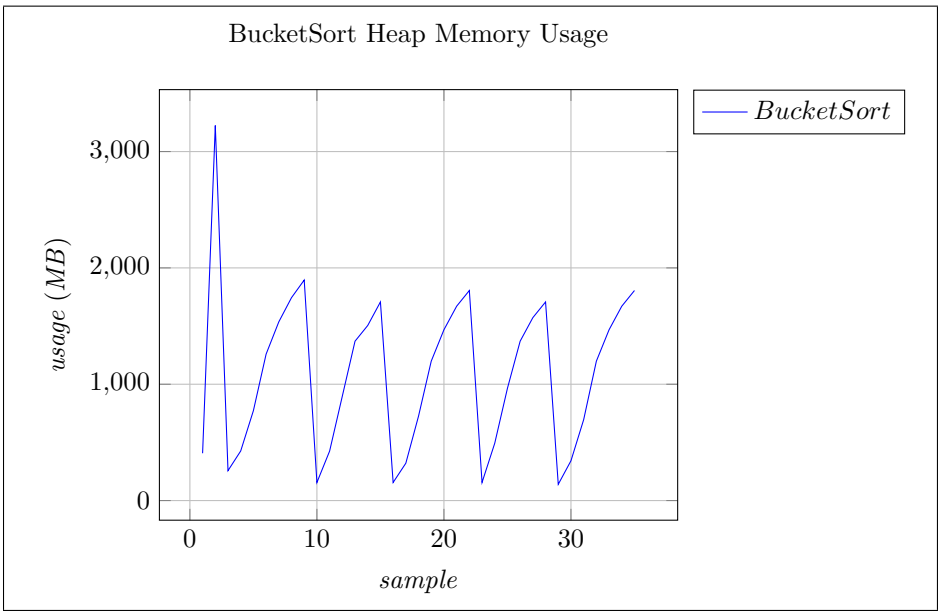

Figure G17: Protected BucketSort heap metrics - 10.000.000 elements.

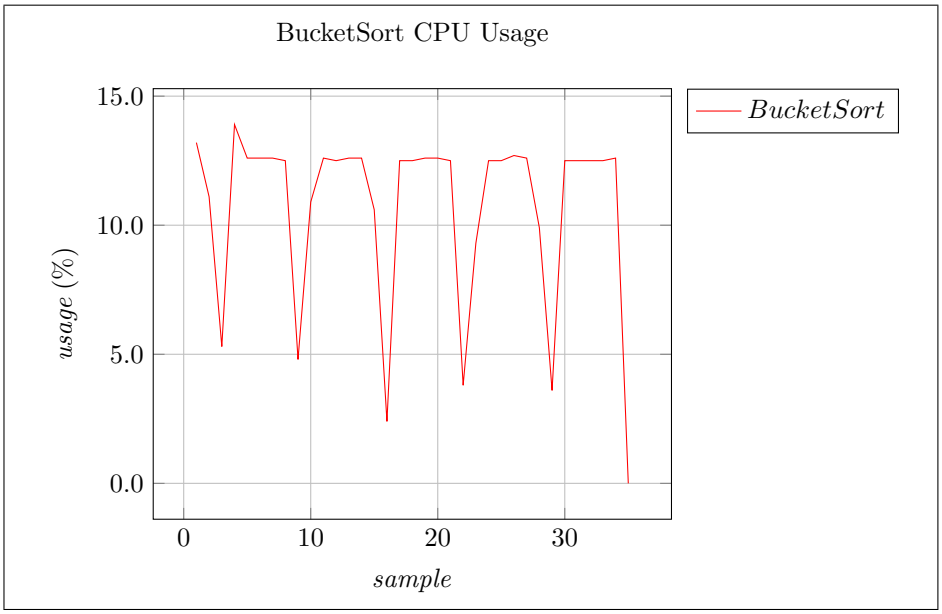

Figure G18: Protected BucketSort CPU metrics - 10.000.000 elements

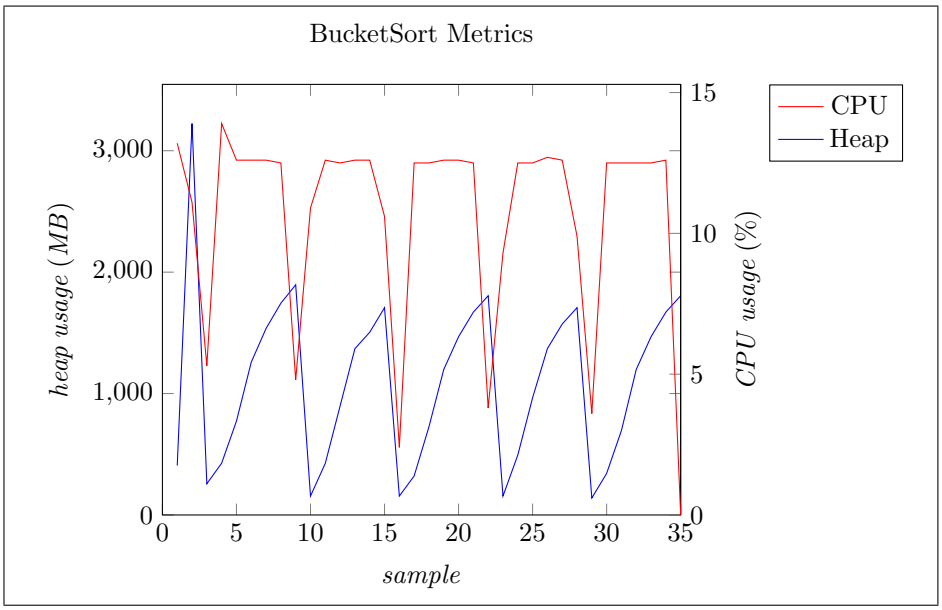

Figure G19: Protected BucketSort metrics combined - 10.000.000 elements.

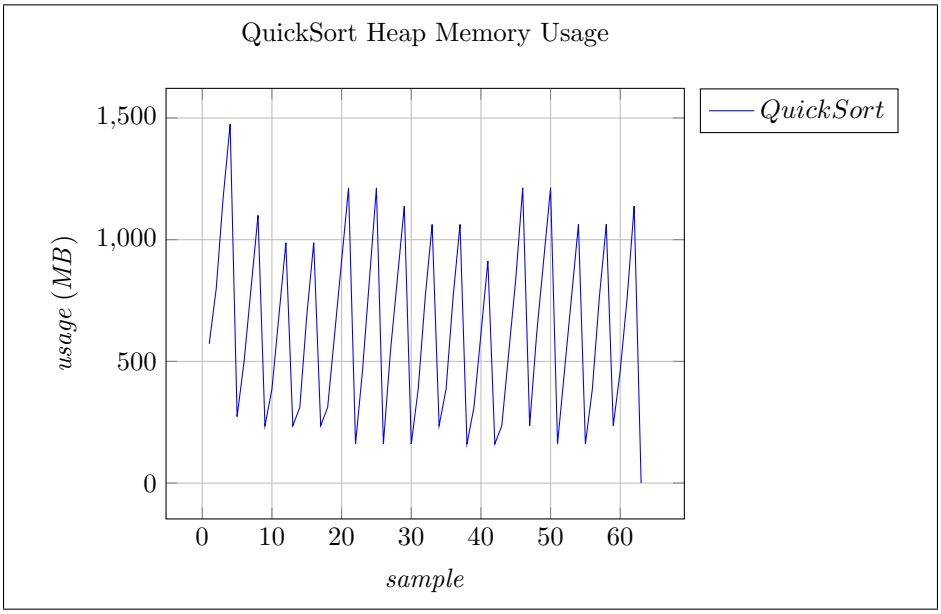

Figure G20: Protected QuickSort heap metrics - 1.000.000 elements.

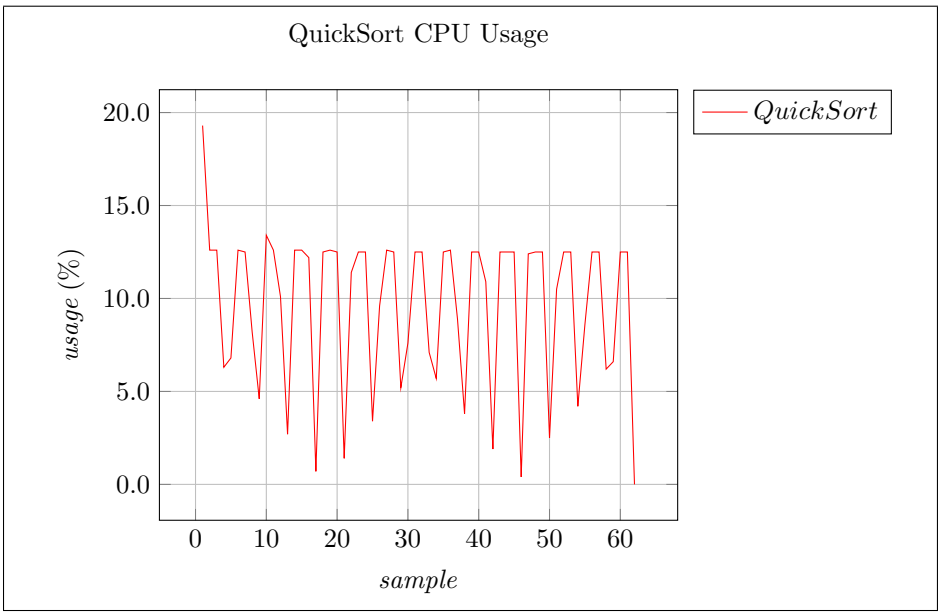

Figure G21: Protected QuickSort CPU metrics - 1.000.000 elements.

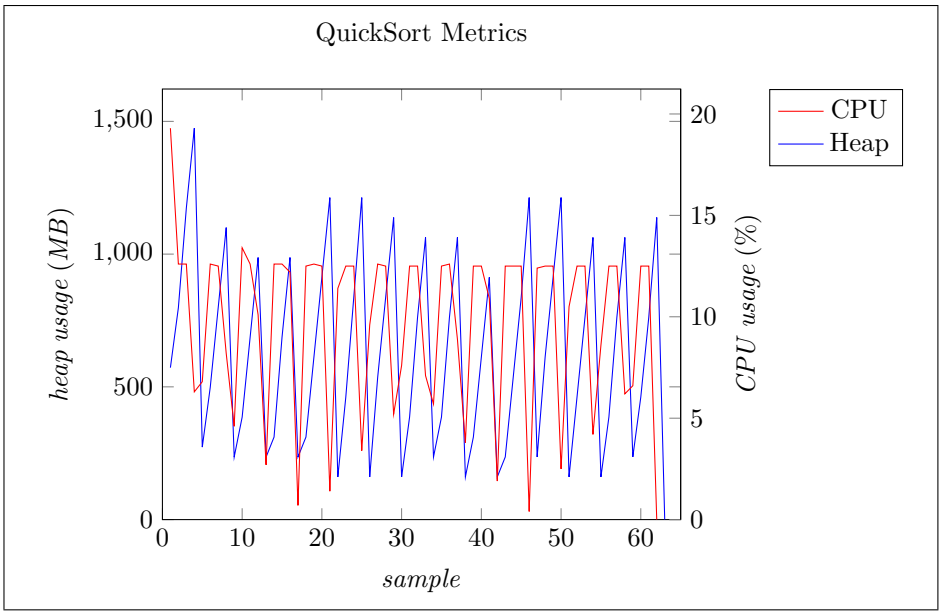

Figure G22: Protected QuickSort combined metrics - 1.000.000 elements.

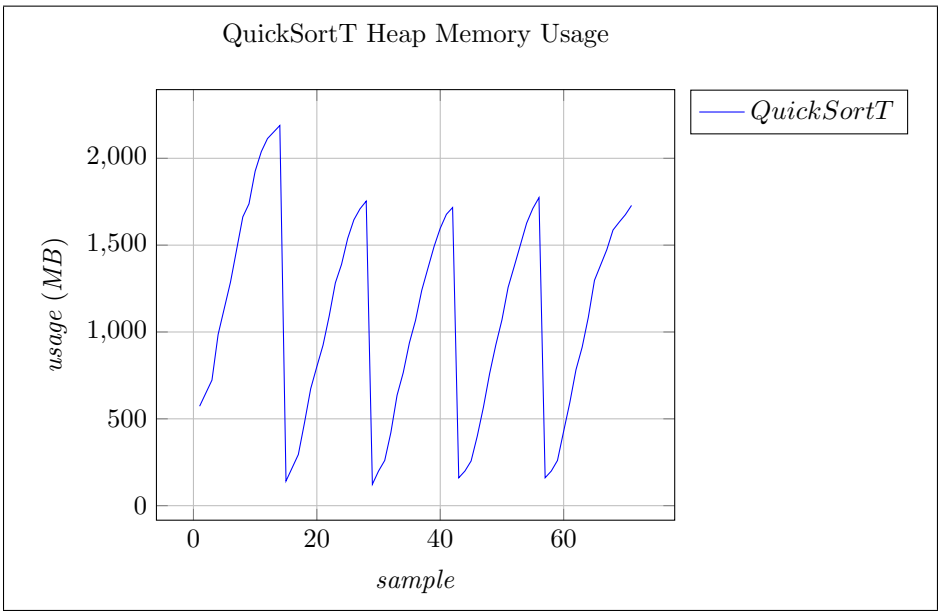

Figure G23: Protected QuickSortT heap metrics - 1.000.000 elements.

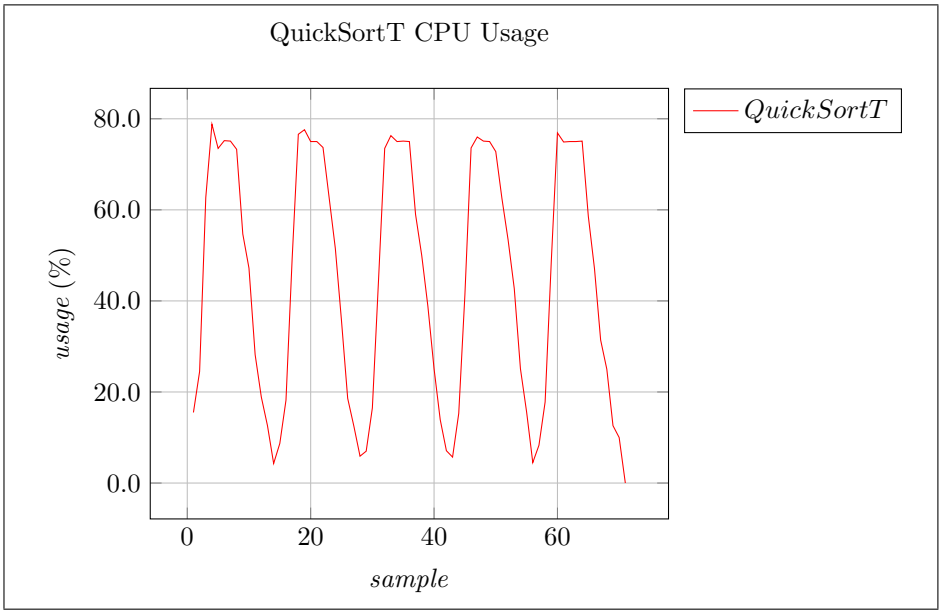

Figure G24: Protected QuickSortT CPU metrics - 1.000.000 elements.

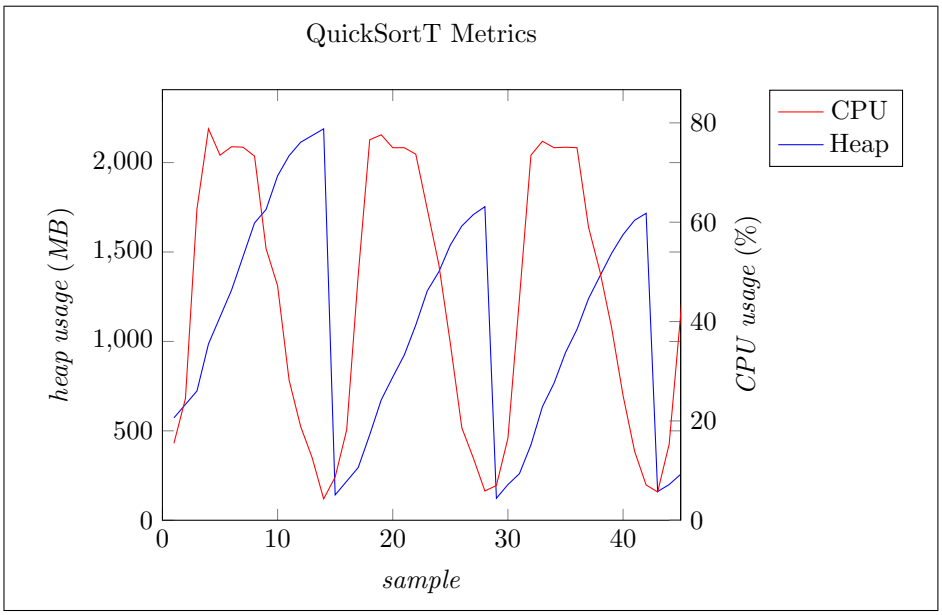

Figure G25: Protected QuickSortT combined metrics - 1.000.000 elements.

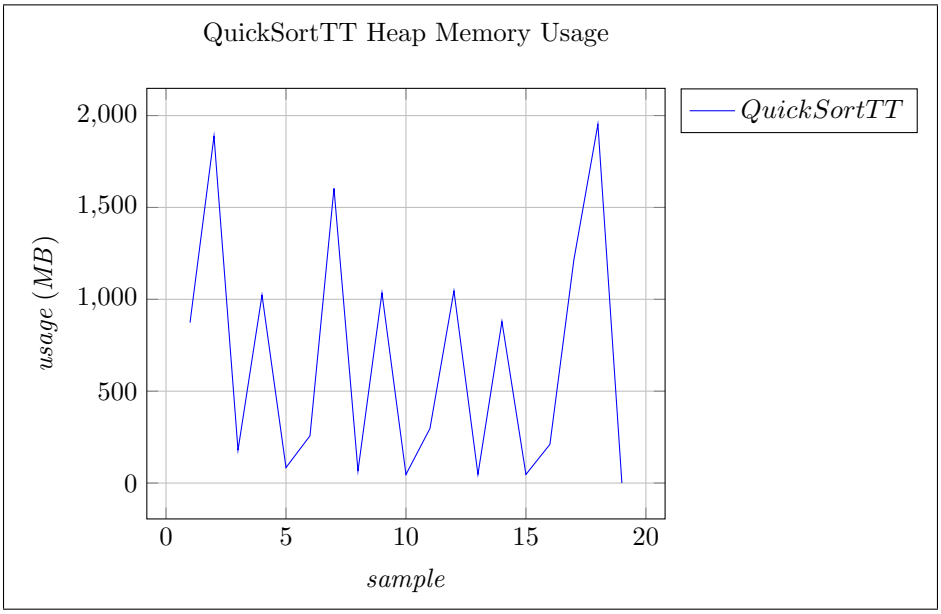

Figure G26: Protected QuickSortTT heap metrics - 1.000.000 elements.

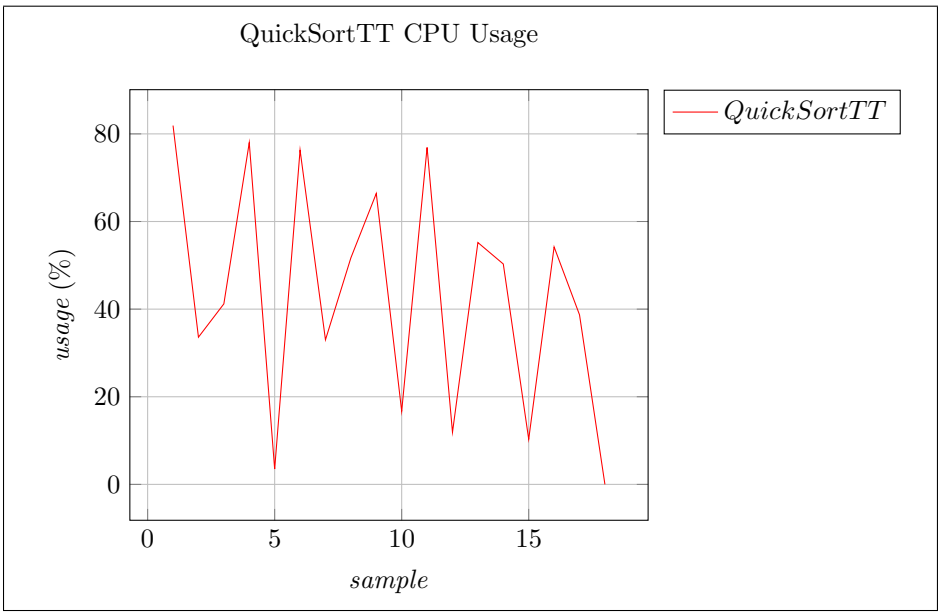

Figure G27: Protected QuickSortTT CPU metrics - 1.000.000 elements.

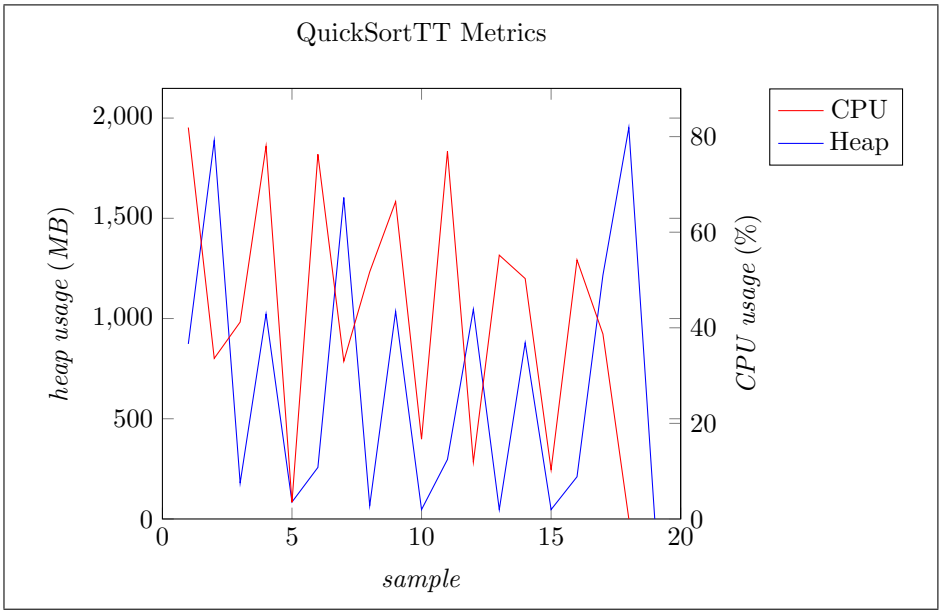

Figure G28: Protected QuickSortTT combined metrics - 1.000.000 elements.

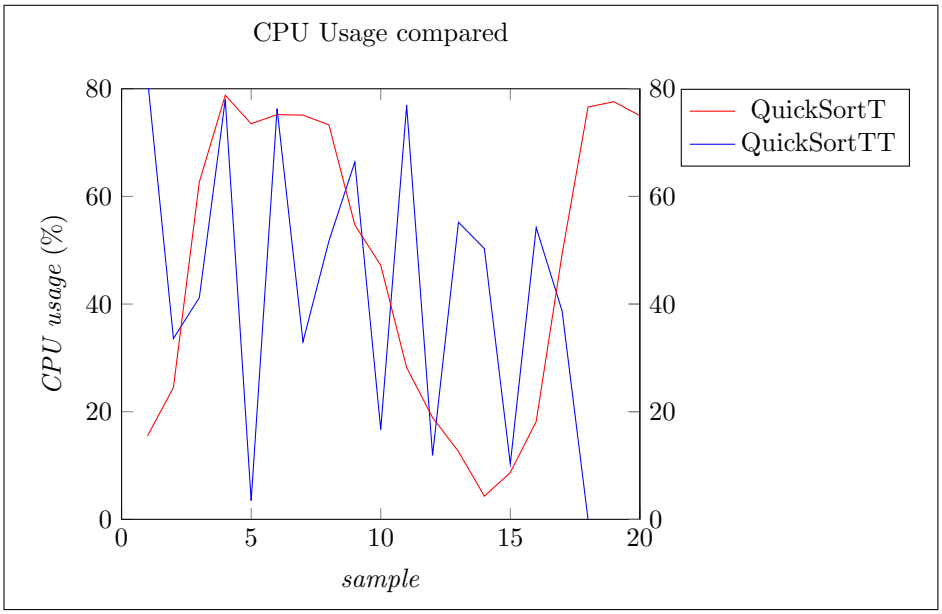

Figure G29: Protected QuickSortT vs QuickSortTT CPU usage - 1.000.000 elements.