JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Dominik Steinbinder

Submitted at
Institute for System Software

Supervisor
Prof. Hanspeter Mössenböck

Co-Supervisor
Dr. Philipp Lengauer

03 2018

Analysis and Visualization of Linux Core Dumps

Master Thesis
to obtain the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

This thesis deals with an automated approach to analyzing Linux core dumps. Core dumps are binary files that contain the runtime memory of a process running in a Linux based operating system. The dump analysis tool extracts and visualizes relevant information from core dumps. More advanced users can start a manual debugging session from within the web interface and thereby access all functions that a conventional debugger supports. By integrating the tool into a third party issue tracking system, core dumps are fetched and analyzed automatically. Therefore, the analysis is started as soon as a core dump is uploaded into the issue tracking system. The tool also keeps track of analysis results. The collected data can be filtered and visualized in order to provide a statistical evaluation but also to detect anomalies in the crash history.

Zusammenfassung

Diese Arbeit befasst sich mit der automatischen Analyse von Linux Core Dumps. Core Dumps sind binäre Dateien, die das Speicherabbild eines Prozesses im Betriebssystem Linux enthalten. Das Analyse Tool extrahiert und visualisiert relevante Information aus Core Dumps. Fortgeschrittene Benutzer können über das Web Interface eine manuelle Debugging Session starten und sämtliche Befehle eines konventionellen Debuggers benutzen. Durch die Integration in ein Issue-Tracking-System kann das Tool selbständig Core Dumps finden. Dadurch kann die Analyse direkt nach dem Hochladen in das Issue-Tracking-System gestartet werden. Das Tool speichert die Ergebnisse der Analysen in einer Datenbank. Die Daten können vom Benutzer gefiltert and visualisiert werden, wodurch eine statistische Auswertung möglich wird. Ebenso können die Daten für die Erkennung von Anomalien im Crash Verlauf benutzt werden.

Contents

1 Introduction

Linux-based operating systems can provide a way to persist the memory of a running process to the filesystem. These files are called core dumps. They contain the writable memory sections of the process, i. e., memory sections that were subject to manipulation by the running process, as well as meta information about the environment. Core dumps can be used to analyze the state of a process at a specific time, e. g., the state immediately before the process crashed. Thus, core dump analysis allows to debug applications post mortem.

The information stored in a core dump is vast because the complete writable memory of the target process is stored in addition to some meta information. Storing this much raw information also comes at a price: core dump analysis without tool support is unfeasible. Fortunately, there are tools that developers can use, so they do not have to bother with low level information but can focus on relevant high level information.

The spectrum of tools allowing post-mortem debugging with core dumps is rather limited. The state-of-the-art tool is the *Gnu Debugger* (GDB), a powerful command-line tool [1]. Originally, it was a debugging tool but was later extended to a post-mortem debugger by supporting core dumps as input instead of running processes.

Although GDB is the most widely used tool, there are some alternatives, e. g., the *DBX debugger* from IBM [2], the *Modular Debugger* from Oracle [3], and the *LLDB debugger* from the LLVM Developer Group [4].

By default, these tools come with a command line interface. However, there are extensions that provide a graphical user interface for GDB, DBX, and LLDB. The functionality of all four tools is rather similar. In fact, there are tables that translate a command for one debugger to another [5] [6] [4]. While these tools have a very large number of functions making them the perfect choice for sophisticated in-depth debugging, they also have drawbacks.

One of these major drawbacks lies in the complexity. Users without prior debugging experience will have troubles understanding the commands. Furthermore, even if the user is experienced in debugging, he must invest a considerate amount of time to learn the individual commands for the specific debugger. Finally, even if the user has debugging experience as well as experience with the specific debugger, the setup process that is required for a fully fledged debugging session will still slow him down.

1.1 Motivation

The tool developed in this thesis should provide an automated way of analyzing core dumps as well as a high-level visualization of the extracted information. Therefore, the disadvantages of conventional debuggers mentioned previously are mitigated. Unexperienced developers can use the provided high-level information while experienced developers can still profit from the quick analysis.

As an immediate result, the tool cannot be as powerful as existing command line tools such as GDB and should not be used as a replacement for these. It should rather assist developers in getting the most important information from a Linux core dump in an automated way. In certain cases, the fallback to conventional debugging tools is inevitable.

1.2 Specification

The goal of this thesis is to provide an automated way of presenting the most important information stored in a Linux core dump. This includes, but is not limited to, general system information, thread information including stack traces of each thread, and a list of loaded libraries. The tool must be integrated into the issue tracking system JIRA to automatically search and analyze Linux core dumps. The results of the analysis shall be presented in a web interface based on that of the Windows core dump analyzer SuperDump [7]. Additionally, the tool must maintain a database of the analysis results and provide a summary about the most common scenarios that lead to a crash.

The thesis must also include a functional evaluation and a performance evaluation. The functional evaluation must prove that the tool can extract the required information based on pre-defined JIRA issues and compare its capabilities to existing tools such as GDB. The performance evaluation must provide some intuition about the run-time and memory requirements of the developed tool.

1.3 Limitations

This thesis is designed to provide an environment-independent way of analyzing core dumps. However, some parts of the work depend on the surrounding environment, e. g., the input for core dumps is required to be an archive with a specific structure. These parts are accordingly indicated and explained.

1.4 Publishing

The tool has been licensed under the MIT license and is published along with SuperDump in the Github repository of Dynatrace: `https://github.com/Dynatrace/superdump`. Although the repository contains code for both Windows and Linux dump analysis, the Linux core dump analysis tool can be deployed as a standalone application. More information on deployment is given in Chapter 5.

2 Background

This chapter describes the environment in which the Linux core dump analysis tool was developed. It further explains the format of input files for the analysis.

2.1 Integration into SuperDump

The tool developed in this thesis is based on the open source tool *SuperDump* [7]. SuperDump is an open source service for automated Windows crash-dump analysis. In addition to the Windows crash-dump analysis tool, SuperDump provides a web user interface that is mostly independent of specific crash dump types which makes it particularly valuable for this thesis. The tool developed in this thesis can be deployed as an extension to SuperDump to enable analysis of Linux core dumps. The integration into SuperDump comes with the benefit of having the user interface already set up. Only minor adaptions have to be made to visualize the data of Linux core dumps. Also, SuperDump handles the data storage for dump files. SuperDump is written in C# using *ASP.NET* with the *Razor* syntax for the web user interface. Some parts of SuperDump run on the .NET core framework while other parts require the full .NET framework.

Nevertheless, the central work of this thesis, the analysis of core dumps, is independent of SuperDump and can be deployed as a stand-alone tool.

2.2 Programming Languages and Frameworks

The analysis part of the developed tool is written in C# for the .NET core framework version 1.1. The major reason for this decision was to assure code consistency across Windows and Linux analysis. Also, parts of the Windows analysis can be reused for processing Linux core dumps. The *libunwind wrapper* described in Section 3.1 is written in C++ because it depends on *libunwind*, a native library for unwinding stacktraces.

2.3 Analysis Inputs

A core dump file only contains the writable memory sections of the process but not the static sections. Restoring the complete memory image requires all loaded binary files. Thus, to do a thorough analysis, the executable file and the shared libraries must be available in addition to the core dump file.

In order to still have only a single file when submitting a core dump, a feature that automatically extracts files from an archive has been implemented. The required archive structure is shown in figure 1. The archive types in the figure denote **.tar.gz* archives but the archive type is interchangeable. Supported archive types are: *.tar, *.tar.gz, *.zip, *.bzip, *.bzip2.

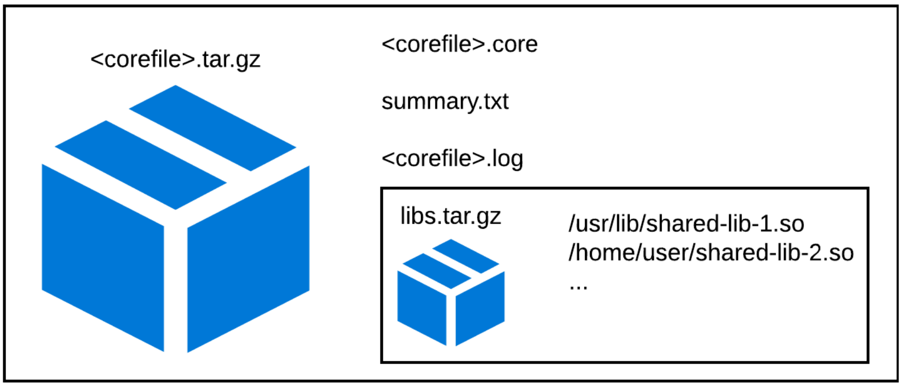


Figure 1: Expected structure of incoming core dumps. The term *<corefile>* is used as a placeholder for the name of the core dump file. The actual core dump file (*<corefile>.core*) and the shared libraries (*libs.tar.gz*) are mandatory for the analysis.

- Shared libraries: A core dump contains the memory sections of the target process, but it does not contain any binary files to help interpret the memory sections. In order to retrieve a meaningful analysis report, the binary file and all shared libraries are required. These files are expected to be in an archive with the original filesystem structure as indicated in figure 1.
- Crash summary file: the core dump itself provides little meta information. A summary file includes relevant meta information about the application, e. g., the path of the executable.
- Log file: In the Dynatrace environment, an additional log file is written when a core dump is generated. This log file contains version information for the shared libraries. Note that this file is specific to the tool that processed the dump file and will not be available by default.

Any files but the core dump itself are not created by default in a Linux operating system. An according tool for dump processing has to be installed to guarantee valid analysis reports. The implementation and installation of such a tool is not in the scope of this thesis. It is assumed that the described files are present in each dump. However, a missing log file merely results in the loss of version information for shared libraries. The crash summary file is only required in a special case where the path of the executable cannot be extracted from the core dump file. This special case will be described in

Section 3.5. Note that shared libraries are required for the major parts of the analysis and must not be missing.

In following chapters, the term *core dump file* will refer to the single uncompressed core dump file, while the term *core dump* will refer to the archive containing the core dump file and any additional files.

2.4 Core Dump Structure

Core dump files must adhere to a common structure because they contain more information than the plain memory of the target process. In particular, they comply to the specification of the *Executable and Linking Format* (ELF specification) [8]. The ELF specification defines three different types of object files: relocatable files, executable files, and shared object files:

- *Relocatable files* contain information that can be used to create executable files or shared object files.

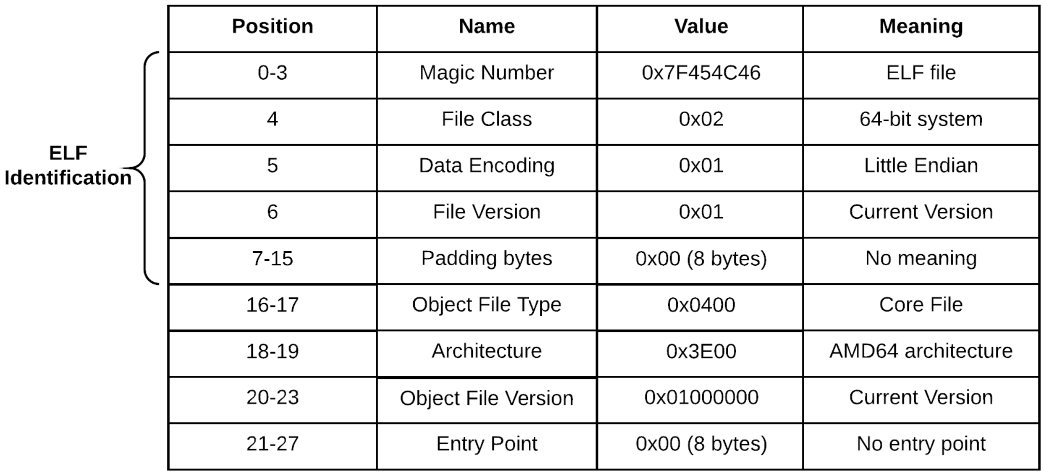
- *Executable files* contain information about the program to be executed, in particular its memory segments and the starting address.

- *Shared object files* can be used among multiple executable files. They contain both code and data.

Core dump files do not need an explicit type declaration because their data fits with the definition of executable files. The primary purpose of both executable files and core dump files is to create a memory image. With executable files, a new memory image is created, whereas with core dump files, the memory image of a process at a certain state is restored. There are two relevant views for showing the contents of an ELF file: the *execution view* and the *linking view.* While the execution view provides information relevant when executing a file, the linking view focuses on information used by linkers. Table 1 shows the *execution view* for ELF files. The linking view is irrelevant for the scope of this thesis.

Execution View
ELF header
Program header table
Segment 1
Segment 2
...
Section header table *optional*

Table 1: The ELF object file format in the execution view is described in the ELF specification [8]


ELF Identification

Position	Name	Value	Meaning
0-3	Magic Number	0x7F454C46	ELF file
4	File Class	0x02	64-bit system
5	Data Encoding	0x01	Little Endian
6	File Version	0x01	Current Version
7-15	Padding bytes	0x00 (8 bytes)	No meaning
16-17	Object File Type	0x0400	Core File
18-19	Architecture	0x3E00	AMD64 architecture
20-23	Object File Version	0x01000000	Current Version
21-27	Entry Point	0x00 (8 bytes)	No entry point

Figure 2: Parts of an example ELF Header

The ELF header describes the organization of the file and provides some information about the target system such as the system's bitness. The precise structure of the ELF header is described in Subsection 2.4.1.

Data in an ELF file can be stored either in *sections* or in *segments*. While sections contain data that is used during linking, segments contain data that is used at run-time. Sections and segments are defined in the section header table and in the program header table, respectively. As segments hold information that is required to create the memory image, the program header table plays an important role in analyzing core dumps. Linking information is irrelevant for core dump files. Thus, they usually do not contain any sections.

2.4.1 ELF Header

This subsection describes the structure of the ELF header. A particular focus lies on data that is relevant for core dump analysis.

Figure 2 shows parts of an ELF header that was extracted from a core dump file. Each entry is briefly described, while a more exhaustive explanation is provided in following paragraphs.

ELF identification The file's contents start with the magic number identifying a file that conforms to the ELF standard. The value of the first 4 bytes is always *0x7F454C46* which in ASCII stands for ".ELF". The next byte declares the object's sizes and therefore also the bitness of the target system: the value is either *1* for a 32-bit architecture or *2* for a 64-bit architecture. This information allows us to immediately distinguish between 32-bit dumps and 64-bit dumps which is essential in the analysis approach. The following byte describes the endianess of the target system: the value is either *1* for little endian or *2* for big endian. The next byte shows the ELF header version number which is *1* by definition. The remaining bytes in the ELF identification are unused. These bytes may be used to provide additional information in future versions.

Object file type These two bytes are used to identify the file type. Table 2 describes the available file types.

Value	**Meaning**
0	No file type
1	Relocatable file
2	Executable file
3	Shared object file
4	Core file
0xFF00 - 0xFFFF	Processor-specific

Table 2: Object file types as described in the ELF specification [8]

With this information, we do not have to rely on the file name ending in *.core* to identify a core dump file but we can use the header information in the first few bytes of a file.

Architecture The value contained in this field identifies the architecture. The identifiers are organized and published by the SCO Group [9].

Object file version The original specification as described here uses version 1. Tools can extend the file format by adding additional information and incrementing the version number. The analysis methods discussed in this thesis do not require any extensions and is therefore able to handle all core dump files that comply to the original ELF specification.

Entry point This field holds the starting address of the process. A value of *0* would indicate that the file is not supposed to be executed.

Further fields The ELF header contains more information about various starting addresses, offsets, and counts. However, they are omitted in this elaboration because they do not contribute to analyzing core dumps.

2.4.2 Program Header

The ELF program header contains information on how to prepare the program for execution. Therefore, only executable files or shared object files contain a program header whereas relocatable files do not because they are not supposed to be executed. The program header has one or many entries that conform to the structure shown in

Table 3. In practice, most entries describe how to map a given segment to the memory image. Other entries contain auxiliary information that might be relevant for program execution [8].

Name	**Description**
Type	Type of entry (described in detail below)
Offset	Offset from the start of the file to the segment
Virtual address	Start address of the segment in virtual memory
Physical address	Start address of the segment in physical memory
File Size	File image size of the segment in bytes
Memory Size	Memory image size of the segment in bytes
Flags	Various flags that are relevant to the segment
Alignment	This value is required for aligning segments in memory and in the file

Table 3: Fields in a program header entry are described in the ELF specification [8]

Potential values for the entry type are listed in Table 4. Program header entries of type *PT_LOAD* are relevant for restoring the memory image which is a prerequisite for analyzing the raw memory of a process. Section 3.2 describes how this information can be used to retrieve the list of shared libraries.

Name	**Description**
PT_NULL	Entry is ignored
PT_LOAD	Entry describes a loadable segment, i. e., the value contains mapping information for a specific segment.
PT_DYNAMIC	Entry contains information for dynamic linking
PT_INTERP	Entry specifies the path of the interpreter (only relevant for executable files)
PT_NOTE	Entry contains auxiliary information
PT_SHLIB	Unspecified
PT_PHDR	Entry describes the program header table
PT_LOPROC	Reserved for process-specific semantics
PT_LOPROC	Reserved for process-specific semantics

Table 4: Potential types for program header entries including a brief description. These types are defined in the ELF specification [8].

2.4.3 Notes

Program header entries with type *PT_NOTE* can be used to specify additional information. The ELF specification defines the structure of a note as shown in Table 5.

Since Linux kernel version 3.6, each core dump contains a note with the name *NT_FILE*. This note can help in restoring the memory image because it holds information about

Name	Description
namesz	Size of the name in bytes
descsz	Size of descriptor in bytes
type	The type is used in addition to the name to identify the notes meaning
name	Name of the note
desc	The note descriptor holds the notes data

Table 5: Structure of program header note entries [8]

Name	Code	Description
SIGILL	4	The instruction pointer is located at an illegal instruction; the instruction pointer is persisted
SIGABRT	6	Abort signal indicates an internal error; SIGABRT can be ignored by the process
SIGKILL	9	Kill signal to terminate the process immediately; SIGKILL cannot be ignored
SIGSEGV	11	Segmentation fault indicates a memory problem, e. g., an access the an illegal memory address; this memory address is persisted
SIGTERM	15	Tells a process to terminate; SIGTERM can be ignored by the process

Table 6: Common signal types including their code and a brief description

the address regions of the executable file as well as all shared libraries that were mapped in memory when the core dump was created. More details on how this note can be used for core dump analysis is given in Section 3.2.

2.5 Unix Signals

Signals in Unix operating systems are a means of communication between running processes. The signal type is identified by a single number. These signal types are defined in the POSIX.1 standard, although newer operating systems have extended these signal definitions [10]. Unfortunately, some operating system manufacturers have introduced newer signal types that collide with other systems. Therefore, interpreting the signal types that are not defined in the POSIX.1 standard requires knowledge about the operating system.

Some signal types also provide a related address. For example, the *SIGILL* signal is sent when detecting an illegal instruction. Knowing the address of the illegal instruction proves useful when debugging. Table 6 shows some of the most common signal types.

2.6 Docker

Docker is a container platform [11]. It can be used as a means to mitigate local machine side effects, i. e., to prevent a common scenario in software development where pieces of code only work on a certain machine. Docker is also used to counter side effects of running applications, e. g., it is possible to run multiple webservers listening on port 80 by creating a port mapping with Docker. Applications running inside a so-called Docker container are not allowed to directly access the host operating system unless explicitly configured. A Docker container can be seen as an instance of a Docker image. The container contains data and files from the operating system and from the underlying application. A Docker container can be executed, i. e., the application inside the container is started, whereas the sole purpose of a Docker image is to have a fast and consistent way of creating containers.

For virtualization, the Windows version of Docker, namely "Docker on Windows", uses Hyper-V containers, i. e., a custom Hyper-V virtual machine is instantiated which can run Linux containers [12]. Unfortunately, when running Docker on a Windows Server 2016 system as is done in the Dynatrace environment, only Windows containers are supported for the time being [13]. A number of reports show that Linux containers can be enabled on a Windows Server 2016 by applying various hacks [14]. However, we try refrain from using unsafe techniques wherever possible. Thus, in the SuperDump environment, we are not able to deploy the Linux analysis on the same Windows machine. A separate machine for dealing with Linux core dumps is required.

2.6.1 Images and Containers

A Docker image consists of multiple layers which can hold arbitrary many files. The first layer is usually reserved for the operating system, e. g., a Ubuntu 16.04 system. Each layer depends on the layers below, i. e., only differences to lower layers need to be stored. Figure 3 shows an example structure of an image.

When querying for a file, Docker checks if the file is present starting with the topmost layer. If this layer does not hold the according file, the layer below is checked. Therefore, higher layers are able to overwrite files from lower layers but can also decide to rely on the files from lower layers without having to store them on their own. A *Dockerfile* represents the layers of an image with a single instruction each. The first instruction must declare the base image. Each following instruction adds a new layer on top of the previous one. Typical commands are *COPY* for copying files from the local filesystem to a layer, *RUN* to execute an application inside previous layers, and *CMD* to specify the default execution behaviour of the image. Listing 1 shows an example Dockerfile.

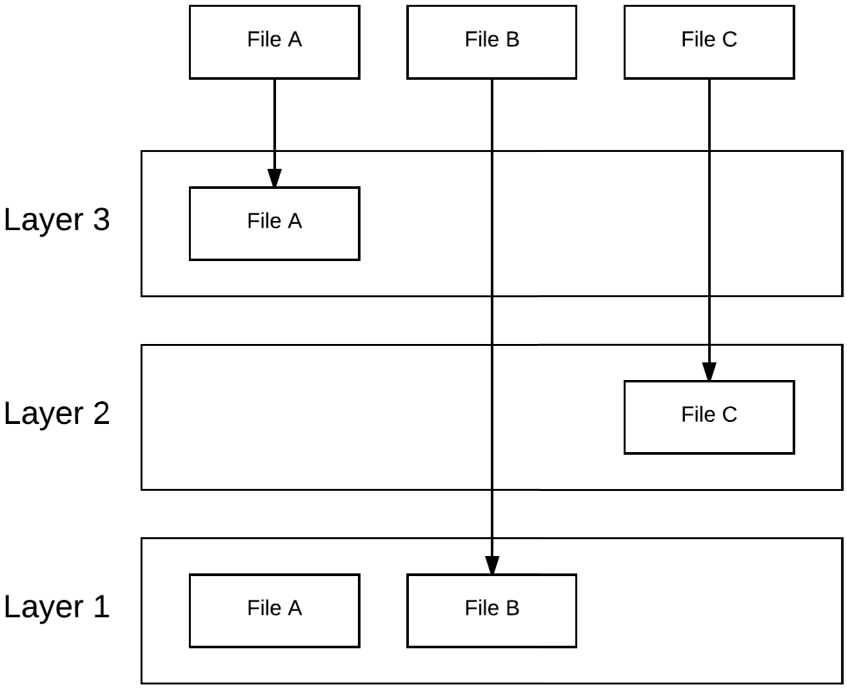


Figure 3: Docker Image Layers. The files A, B, and C are retrieved from layer 3, layer 1, and layer 2, repsectively. The topmost row of the figure shows the user's view on the filesystem when using the container.

```
FROM ubuntu
COPY . /my-app
RUN apt-get update && \
  apt-get install -y gdb
CMD gdb
```

Listing 1: Sample Dockerfile. This Dockerfile uses the most recent Ubuntu version as its first layer, copies the contents from the local directory to */my-app*, installs GDB in the next layer, and sets ”gdb” to be the default execution command if no other command was specified.

In order to execute commands from an image, a Docker container has to be created first. Docker can create a container from an image by adding a writable layer on top of the image layers. This topmost layer is specific to the container and is not shared, while all layers from the image can be shared among multiple containers and images. Any changes made during container execution are written in this writable layer. Thus, instantiating a Docker container is very fast and consumes little disk space. The recommended procedure is to create a new container for each program execution.

2.6.2 Docker Compose

The same way Docker containers can run applications isolated from each other, Docker Compose is able to run multiple Docker containers connected to each other but isolated as an environment [15]. This way, a single command can launch a whole set of containers that share a common network. For the context of Compose, Docker defines the term *service* as the configuration for one particular container. The configuration for all services of an environment is stored in a separate *compose file* which defines the properties of the services as well as how they connect to each other. A sample compose file is shown in Listing 2.

```
version: '2'

services:
  database:
    image: redis:4.0
  frontend:
    image: httpd:2.4
    ports:
      - 8080:80
    volumes:
      - /home/user/my-page/htdocs:/var/www/html
```

Listing 2: Sample Docker Compose file. The core aspect of compose files are service definitions. This script file defines two services: *database* and *frontend*. While the database service is only defined by an image, the frontend service also comes with port forwarding and volume mounting. Port 80 of the instantiated *httpd* containers will be mapped to port 8080 on the host system, while the path */var/www/html* of the container will refer to */home/user/my-page/htdocs* on the host system. The database service does not need to define open ports because the two services share the same network.

2.7 Elasticsearch

Elasticsearch is a search and analytics engine that excels at running queries on large data sets [16]. Elasticsearch aims at a different use case than relational databases: While relational databases are primarily used to store, evaluate, and alter data in tables, Elasticsearch is used for analyzing the data stored in indices. Thus, there are some limitations for getting the most benefits out of Elasticsearch: (1) data in Elasticsearch should only be inserted but never changed or deleted, (2) the data inserted into Elasticsearch must already be evaluated, i. e., no operations should be executed on the data stored, and (3) index joins should be prevented because tool support for joins is rather limited.

While there would be the possibility to use Elasticsearch as the only data storage, it is generally discouraged to do so. The main reason for this is that analyzing and visualizing data is inherently different from merely accessing data. These two distinct use cases demand different storage properties which cannot be satisfied in a single Elasticsearch index. A high grade of redundancy would be required which in turn eradicates the advantages of a single data storage. Therefore, text files serve for accessing dump reports shown in the user interface and Elasticsearch serves as the source for analyzing and visualizing dump data.

3 Analysis of Core Dumps

The analysis of core dumps relies on two types of information: (1) information that is directly retrieved from various parts of the core dump and (2) information that can only be retrieved by inspecting the memory of the target process. While the former can be read from the core dump file, the latter can only be extracted after restoring the memory image which requires all loaded binary files in addition to the core dump file.

This chapter explains how information is retrieved from core dump files. Each section resembles an extraction step with a specific result. The results will be combined to the final report in the end. While most of the extraction steps work independently, some of them depend on information that was retrieved in another step. The order in which the methods are listed reflects these dependencies.

3.1 Unwinding Stacktraces

Stacktraces are important information when it comes to dump analysis. In many cases, stacktraces can indicate the cause of a crash. However, stacktrace information is not directly available but has to be extracted from the memory image which is called *unwinding*. Fortunately, there already exists a solution for this matter, namely *libunwind*.

The library libunwind is able to determine stacktraces of a given program [17]. A major advantage of libunwind is its platform independence through a feature called *remote unwinding*. This feature enables unwinding stacktraces across platforms by allowing a remote process to be the unwinding target. In order to use remote unwinding, an according address space must be created on the local machine that communicates with the remote process.

In the context of core dump analysis, the remote process is replaced with the core dump file. However, libunwind was primarily written to be used on live processes as opposed to core dump files. Core dump support was added later on in version 1.1 released in 2012 [18] and remains rather undocumented. There are some problems and limitations when unwinding stacktraces from a core dump, which are described in following paragraphs. These shortcomings have been mitigated in a custom version of libunwind developed by Dynatrace. This custom version is published in the SuperDump Github repository as a number of patch files that can be applied to the original libunwind source. Besides several bug fixes, some libunwind improvement also add functionality.

Backing files from file notes Libunwind requires a complete list of shared libraries so it is able to access addresses that are not in the address range of the executable file. The easiest way to get this information is to access the *NT_FILE* note which has been added

Name	Description
ENTRY_POINT	Entry point of the program
UID	User ID of the user executing the process
EUID	Effective User ID
GID	Group ID of the user executing the process
EGID	Effective Group ID
PLATFORM	Platform Identification
HWCAP	Machine dependent information
BASE_PLATFORM	Identification for the real platform (may differ from *PLATFORM*)
EXECFN	Path of the executable file

Table 7: Fields stored in the auxiliary vector as described in the getauxv manual [19]

to core dumps since Linux kernel version 3.6. This note stores the addresses with the according file paths. An extension to libunwind has been implemented for automatically retrieving this information.

Signal information The note section of a core dump contains signal information describing which thread received which signal. This information can be useful to determine why a program crashed. For example, the SIGILL signal indicates a problem with the instruction that is supposed to be executed. Further, the signal information indicates which thread caused the termination because in most cases there is only one thread that received a terminating signal. More details on signal information are provided in the corresponding Section 3.6.

Auxiliary vector The auxiliary vector is a note that contains several pieces of information, such as page size, user and group id of the user that executed the process, and more. Table 7 shows the most important fields that get extracted from the auxiliary vector. Please note, that the fields in the auxiliary vector can only be accessed with an index and not by name. Some fields have common indices in all Linux-based operating systems, whereas others differ or are not present in some distributions at all. Therefore, there is no guarantee that the information extracted in this step is always accurate. The patched version of libunwind is able to extract the auxiliary vector.

PRPSINFO The *PRPSINFO* note contains information similar to the auxiliary vector. The reasons why this note is extracted is that it provides information about the process execution, i. e., the path of the executable and the parameters of the call. However, the string containing the complete call information is limited to 80 characters which means that the path or the parameters can get truncated. Any parts of the call information exceeding 80 characters are not contained in a core dump. This constraint becomes especially critical when trying to find the executable file from which the core dump originated. The problem of retrieving the correct executable file is discussed in detail in Section 3.5. The libunwind patch provided by Dynatrace adds the functionality to extract the PRPSINFO note from a core dump file.

Libunwind was written in C for good reasons, such as platform independence and low-level operations. The downside is, however, that using the library in a C# project requires some additional effort. Two potential solutions to this problem were discussed: (1) use *platform invoke* (PInvoke) to directly call libunwind or (2) implement a wrapper in C or C++ that directly calls libunwind functions and provides a simple API tailored to the needs of the core dump analysis tool. As the API of libunwind is rather large and complex, and PInvoke would require to implement a considerable number of data structures and mappings, the chosen solution was to implement a libunwind wrapper. Only relevant functions are implemented in this wrapper and the public wrapper functions are as simple as possible. These wrapper functions are then invoked using PInvoke but there are no additonal interfaces because the wrapper relies on primitive data types which do not require custom interfaces or mappings.

3.2 Shared Libraries

Most of the information that is retrieved for core dump analysis is read from the memory image. The core dump file contains details required to restore the memory image but it does not contain any binaries. Instead, it holds the mounting addresses and paths of the original binary files that were used when the process was running. Without having the binary files, the memory image cannot be restored and dump analysis will be limited to information that is retrieved directly from the core dump. This section describes two approaches for finding out the addresses and paths of the loaded binary files. Both approaches have been implemented as each of them provides distinct advantages.

The first approach relies on a specific note. As mentioned in Section 2.4.3, Linux distributions based on a kernel with version 3.6 or higher, store a note with the name *NT_FILE* in each dump [20]. This note contains the mounting addresses as well as the local paths of the binaries. Assuming that the original path of the binary files at the time of process execution is available to the core dump analysis tool, this information suffices to restore the memory image. Section 3.1 described the extension for libunwind to be able to read this note and automatically map relevant memory segments. This approach is more convenient than others because all required information can be extracted from the note section.

Nevertheless, dumps that were generated on Linux operating systems with a kernel version lower than 3.6 cannot be analyzed with this method. At the time of writing, several Linux distribution still use lower kernel versions, e. g., Red Hat Enterprise Linux 6.9 released in March 2017 uses Linux kernel version 2.6.32. The kernel shipped with these releases is not the original Linux kernel but has been modified to include important security and bug fixes from newer versions. Unfortunately, the patch for including the *NT_FILE* note was not included in several distributions, e. g., the Red Hat Linux distributions.

Therefore, the second approach extracts the required information from the *PT_LOAD* entries in the program header. However, this information does not include any file paths which are crucial for finding the binary files on the filesystem. The procedure of retrieving these file paths is thus rather complex and a detailed description would exceed the scope of this thesis. Fortunately, with the help of two tools, the mapping information can be extracted from the program header: the debugger GDB and the Unix tool *readelf* that translates the binary contents of an ELF file to human readable output.

Figure 4 shows the memory region of a shared library in a memory image. In order to restore the memory region for the library, the library file needs to be mapped to the base address of the shared library in the memory image. First, the *info sharedlibrary* command of GDB is invoked to get the addresses that are denoted in program header entries of type *PT_LOAD*. The addresses extracted hereby point to the start of the *.text* section of the referenced file. As the complete memory of the shared library needs to be mapped, the base address of the shared library is required. The tool readelf is able to show the offset of the *.text* section from the base address. Listing 3 shows a condensed output of a readelf invocation. Readelf is used to extract the offset of the *.text* section. Invoking readelf with the argument *-S* prints only section data. The *.text* section offset is shown in the column labeled *Off* of the 11th section header, namely the *.text* section. Subtracting this offset from the given address yields the correct base address.

```
> readelf -S libperl.so.5.18
There are 27 section headers, starting at offset 0x188398:

Section-Header:
  [Nr] Name              Type             Address          Off
       Size              ES               Flg   Lk   Inf   Al
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000         0     0     0
  [ 1] .note.gnu.build-i NOTE             00000000000001c8  000001c8
       0000000000000024  0000000000000000   A     0     0     4
  ...
  [11] .text             PROGBITS         0000000000029960  00029960
       00000000001198b5  0000000000000000  AX     0     0     16
  [12] .fini             PROGBITS         0000000000143218  00143218
       0000000000000009  0000000000000000  AX     0     0     4
  ...
```

Listing 3: Using readelf to extract the *.text* section offset

3.3 Debug Symbols

Debug symbols contain information relevant for debugging, e. g., variable and method names. Most compilers support both storage options for debug symbols: either in the binary file or in a separate debug symbol file. Adding debug symbols to the binary file has several drawbacks, such as the added space requirements for the binary file and the exposure of sensitive information to any user that can acquire the binary file. Therefore, the common approach is to create separate debug symbol files and make them accessible for developers only.

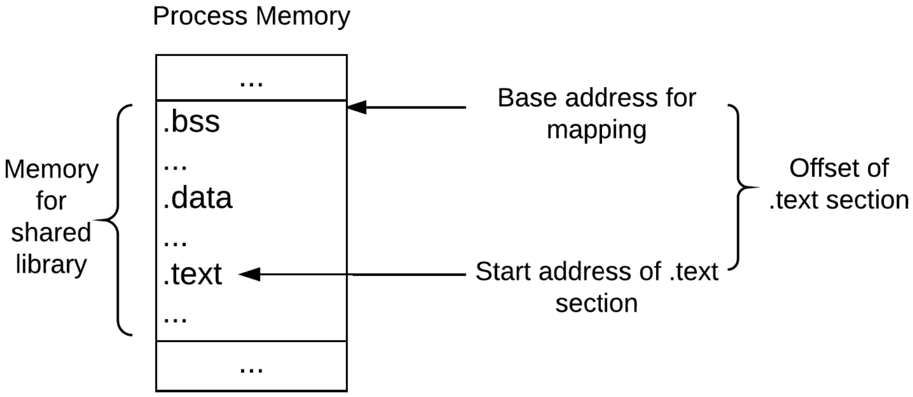


Figure 4: Memory region of a shared library.

The implementation for the retrieval of debug symbol files is specific to the Dynatrace environment. Figure 5 shows how debug symbol files are retrieved and processed. At Dynatrace, debug symbol files can be accessed from within the internal network via an HTTP request. The caller has to provide the name of the requested debug symbol file and the MD5 hash of the binary file. In order to get the debug symbol filename, the extension of the binary file must be replaced with ".dbg". If it has no extension, e. g., Linux executable files usually do not have an extension, ".dbg" is appended at the end.

When applying this approach in practice, there are two downsides: (1) the analysis time increases because the requests for the debug symbol files take a long time in comparison to the actual analysis time and (2) debug symbol files are only stored for several days on the debug symbol store at Dynatrace, i. e., the tool will fail to provide debug symbols for older dumps. To circumvent these two problems, a cache has been implemented that stores the debug symbol files on the local disk such that they are accessible with their hash and debug symbol filename. However, a cache introduces the potential problem of using large amounts of disk space for symbol files that are no longer needed. A retention policy could solve this problem but was not implemented because, at Dynatrace, the cache hit rate is very high and therefore disk space consumption is too low to pose an actual problem.

After debug symbols are available in the local filesystem, we need a way to integrate them into the analysis. The most efficient way would be to reference them in each analysis step that can take advantage of debug symbols. Unfortunately, libunwind in its current implementation is not able to access external debug symbol files. Thus, the debug symbols have to be integrated into the binary files. The *elfutils* package contains a tool named *eu-unstrip* which is capable of merging the debug symbols into the binary files. This process is called *unstripping*, whereas removing the debug symbols from a binary file is called *stripping*.

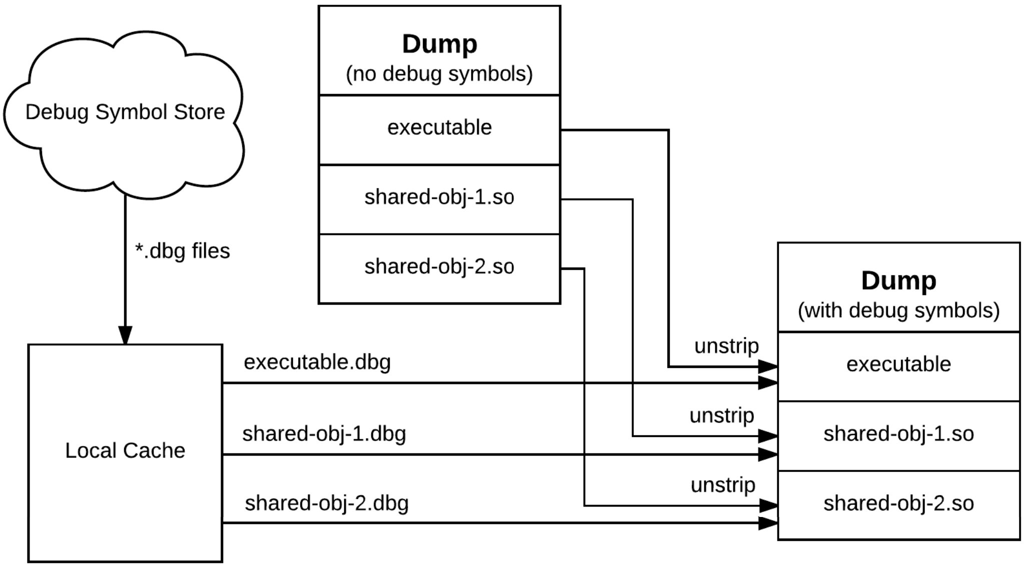


Figure 5: Processing debug symbol files

The downside of this method is the increased disk space requirement as the size of each dump is increased with unstripping. After unstripping a binary file, the binary file's size is increased by the size of the debug symbols. However, Chapter 5 describes how this problem can be mitigated by using a copy of the original dump for analysis.

3.3.1 Source Information

Source information can be especially useful for developers trying to reproduce a crash. For example, if a crash was caused by the access to an illegal memory address, developers can immediately find the violating statement by looking up the exact position in the source code.

Resolving Instruction Addresses into Source Files

Having debug symbols available allows us to resolve instruction addresses to positions within the source code. The Unix tool *addr2line* from the *binutils* package is able to take a binary file and a memory address as input and return the source filepath, the according line number of the statement, and the method name [21]. The invocation command for this use case is based on the pattern: *addr2line -f -C -e <binary-file> <instr-address>*. The flag *-f* tells addr2line to also print out the method name. *-e <binary-file>* specifies the path to the binary file that must also contain debug symbols. As compilers mangle function names, i. e., they modify function names such that they become unique, they have to be demangled. Demangling function names can be enabled in addr2line with the *-C* flag. Optionally, the demangle style can be set along with the flag because compilers may have different mangling styles. Listing 4 shows an example invocation of addr2line and its output. The first line of the output represents the method name while the second line represents the source filepath and the line number, divided by a colon. If debug symbols are not available, or source information cannot be retrieved due to some other reason, question marks are printed instead.

```
> addr2line -f -C -e usr/sbin/varnishd 0x43476C
pan_backtrace
/builddir/varnishd/cache/cache_panic.c:441
```

Listing 4: Invocation of addr2line to retrieve the source filepath, line number, and method name. The address 0x43476C refers to line 441 in the function *pan_backtrace* of the C file *cache_panic.c*.

Resolving Source Files into Repository Paths

In addition to extracting plain source information, the core dump analysis tool is also able to retrieve source files for the linked binaries from a repository. Having the source files present on the local filesystem proves useful for the interactive mode which will be described in Section 3.8. As the retrieval of source files from the repository is highly specific to Dynatrace, this part will only be explained briefly here.

The source paths that have been extracted with addr2line are the paths that the compiler used when generating the executable file. However, in order to download source files from the repository, the repository paths of the source file are required. The according path transformation is highly specific to the environment and depends on the implicit information stored in the original source file path as well as the repository structure. In the Dynatrace environment, the complete repository path can be generated with little effort but there is no way of finding out the version of the source file. However, the original source file paths contain the Sprint version in which the sources were compiled. The Sprint version narrows the time frame down to the length of a Sprint, i. e., two weeks. The tool fetches the source files that were available at the end of the according Sprint. Although this procedure does not yield perfect results, it is as accurate as possible with the information given.

3.3.2 Stack Information

Debug symbols can also be used to retrieve stack information, in particular to retrieve the names and values of function arguments and local variables. This additional information can assist developers in tracing back the crash of an application.

However, retrieving stack information is a rather complex procedure as it requires analyzing the debug symbols to restore the stack structure at the given instruction address. When the structure of the stack is known, the stack variables can be read from the restored memory image of the dump. Stack variables are accessed by subtracting the according offsets from the original stack pointer and accessing that address in the restored memory image. Fortunately, GDB already has this functionality implemented in its commands *info args* and *info locals* printing out function arguments and local variables, respectively. Listing 5 shows how to use GDB to access this information.

```
> gdb /usr/sbin/varnishd crash-varnish.core
> (gdb) thread 2
> (gdb) frame 3
> info args
this = 0x7faf3cc440a0
size = 32
> info locals
aws = 0x7faf46c5a3e0 "\345B!/"
```

Listing 5: Accessing stack information with GDB. The commands *thread 2* and *frame 3* select the third frame of the second thread. The invocations *info args* and *info locals* shows the names and values of three stack variables, namely *this*, *size*, and *aws*. The local variable *aws* refers to a character array and therefore, the string contents will be displayed.

Function arguments and local variables are not the only information that can be retrieved from the stack. For example, global variables could be retrieved using the GDB commands *info variables* and *print <variable-name>*. However, practice has shown that the amount of global variables for real-world applications is vast and the provided information does not help with debugging in most cases. Users can still access this info by using the interactive mode which will be described in Section 3.8.

3.4 Analysis Report Tags

Upon inspecting the analysis report of several core dumps, distinct patterns are emerging. Some of these patterns indicate a concrete problem and should be highlighted to provide a faster workflow for users.

With the *tagging* functionality, predefined tags can be added to various analysis report elements, such as stacktrace lines or complete threads. Each tag needs to define an element type and a rule for matching. For example, the *ClrWaitForGc* tag states that the method name of a stacktrace line must contain “GCHeap::WaitUntilGCComplete”. When a stacktrace matches the rule, the stacktrace line gets “tagged”, i. e., the name of the tag is stored in the analysis report along with the stacktrace line. The user interface can then highlight this stacktrace line. In the Dynatrace environment, the main purpose of this feature is to highlight stacktrace lines that are related to products developed by Dynatrace. Therefore, users of the analysis reports can immediately see which parts of the stacktraces are related to a specific product. Nevertheless, tagging can also be applied to shared libraries and threads.

Tagging is a feature that was already introduced in SuperDump for Windows crash dumps. In the scope of this thesis, the functionality has been adapted to support Linux core dumps.

3.5 Executable and Call Arguments

Retrieving the executable's file path is crucial for further analysis. As described in Section 3.2, all binary files including the executable file must be mounted in order to analyze the core dump. Unfortunately, extracting the executable's file path in a reliable way is not as easy as it would seem. Following paragraphs discuss three methods to retrieve the file path of the executable file.

Notes Section 3.1 stated that the *PRPSINFO* note in the notes section contains auxiliary information. Within this auxiliary information, there are two relevant fields: the file name and the arguments. These fields are character arrays limited to 16 and 80 characters, respectively. The file name does not contain any path information and will therefore only be stored in the analysis report without further processing. On the other hand, the argument field holds the complete file path including the arguments that were used when starting the process. Using the argument field to find the executable proved as a valid approach and yielded good results. However, for a few core dumps, 80 characters did not suffice to store the complete file path. In such cases, the file path got truncated after the 80th character.

Third party files The most reliable solution is to store the executable's file path in a separate file when creating a core dump. The downside of this approach is that it requires additional setup on the client side, i. e., the tool that creates the core dump. In the Dynatrace environment, the information is stored in a file “summary.txt”.

Heuristic analysis of present files Another approach is to iterate over all files that are provided with the dump and filter for executable files. The filter checks if a certain file is an executable file according to the ELF specification as described in Section 2.4.1. The downside of this approach is that it may produce wrong results if the submitted core dump, for any reason, contains multiple executables. Therefore, it was not implemented in the final version of the dump analysis tool.

The tool developed in this thesis implements the first two methods that were described to retrieve the executable's file path. The primary source is the third party file “summary.txt” as it proves more accurate than other solutions. If the third party file does not exist or does not contain the required information, the argument field in the *PRPSINFO* note is queried as described above. But this approach might not be successful if the file path exceeds the limit of 80 characters.

In conclusion, there is no way to guarantee a correct retrieval of the executable's file path without having a third party file. Additional fallback methods could have been implemented but were omitted as practice has shown very good results.

3.6 Signal Information

Signals are often used to terminate a process, e. g., sending the *SIGKILL* signal to a process shuts down the process immediately. Therefore, signal information gives insight on why the program crashed or terminated in the first place. Signal information is stored in ELF notes with the name *NT_SIGINFO* on a per-thread basis. Thus, extracting signal information not only lets us identify the type of signal, but we can also infer which thread caused the termination because only one thread will receive a terminating signal.

3.7 Additional Information stored in Third Party Files

Not only the core dump file and the attached binary files can be used for analysis but also third party files such as log files can contain relevant information. However, these third party files are not generated by default.

In the Dynatrace environment, the file path of the executable and the version information of certain shared libraries can be retrieved from third party text files.

3.7.1 Module Versions

While the file path of the executable is required for further analysis, module version information is solely stored in the analysis report. A file with the extension ".log" stores version information about binaries that were developed within the Dynatrace company. This information proves useful for finding the error cause. An example extract from such a log file is shown in Listing 6.

```
mapped files:
  7f1355187000-7f1355207000  0 /lib/liboneagentproc.so (1.115.0.20170213-130951)
  7f1355207000-7f1355407000 80 /lib/liboneagentproc.so (1.115.0.20170213-130951)
  7f1355407000-7f135540a000 80 /lib/liboneagentproc.so (1.115.0.20170213-130951)
  7f135540a000-7f135540b000 83 /lib/liboneagentproc.so (1.115.0.20170213-130951)
```

Listing 6: Extracting library information from a third party log file. The version is located inside the parentheses after the file path of the binary file.

3.7.2 Executable Location

As mentioned in Section 3.5, the file path of the executable file is stored in a file with the name "summary.txt". The actual file path can be retrieved with a simple regular expression. An example of the contents in the "summary.txt" file is shown in Listing 7.

```
executablePath: /home/tux/binaries/linux-x86_64-debug/libagent-test
faultLocation: libagent-test!util::kaboom()+0x1c
faultModulePath: /home/tux/binaries/linux-x86_64-debug/libagent-test
size: 54 MB
```

Listing 7: Extracting the path of the executable file from the "summary.txt" file. The executable path is placed in the first line of the file's contents.

3.8 Interactive Mode

As explained in the introduction, the goal of this thesis is not to provide a substitute for in-depth debugging tools such as GDB. For certain core dumps, the use of these low-level tools is inevitable. The interactive mode is an attempt to make in-depth debugging faster and more convenient for users.

The interactive mode allows users of the web interface to create a live debugging session with an in-depth debugging tool. Users can benefit in two ways: first, they have access to a Linux debugging tool within their web browser, i. e., they can use GDB even on a Windows operating system and second, the setup effort for each individual dump can be automated minimizing the manual effort for the user. The specific setup steps are as follows:

1. The core dump file and the executable are loaded in GDB. This way, the user does not have to look for the correct file paths.
2. Shared libraries are automatically made available to GDB. These files enable GDB to display more details about the core dump.
3. If source files of the binaries are available in a code repository, they are made accessible for GDB. GDB will display parts of the source code when viewing according stacktrace frames.
4. An arbitrary additional command makes usage of the interactive mode faster, e. g., the user interface can redirect the user to an interactive debugging session that will automatically show information about the current thread.

Performing these setup steps manually would require a considerable amount of time even for very experienced developers. With the interactive mode, a fully fledged debugging session is available within a few seconds and no manual user effort. The knowledge of using GDB is still required, however.

3.8.1 Implementation

The imlementation choices for the interactive mode have been made on basis of the requirements which were discussed with the Dynatrace team as follows:

1. Usability: Most developers that do in-depth core dump debugging are used to terminal windows. In order to make the interactive mode a viable replacement to the conventional approach of starting a terminal with a debugging tool, the interactive mode must provide a web interface that is similar to actual terminals. Having a terminal in a web interface may seem unplausible at first but is the only viable way in the scope of this thesis to enable a GDB debugging session within a browser.
2. Setup: Most of the required setup steps for GDB should be taken care of automatically. This is one of the main advantages of the interactive mode.

3. Gnu Debugger: The chosen debugging tool for the interactive mode was GDB because most developers working at Dynatrace were already familiar with it.

4. The webserver should not be tightly coupled with the machine running the core dump analysis. Several advantages follow from this restriction. First, the webserver is scalable independently of the analysis machine. Second, the webserver can be exchanged or updated without affecting the analysis machine. And lastly, failures on behalf of the webserver are unlikely to affect the analysis machine. A downtime of the interactive mode is not as critical as an outage of the complete core dump analysis service.

Following from these requirements, several conclusion can be made. First, in order to provide a highly usable web terminal, a TTY interface was chosen for communicating with the user. For example, a TTY interface allows the user to scroll the command history by using up and down arrow keys. Also, auto-completion of commands with the tab key is possible. Luckily, there are webserver implementations that provide a terminal with TTY support. The tool *GoTTY* developed by Iwasaki Yudai was chosen as the tool of choice because it provides all necessary functions, is easy to install and use, and comes with the MIT license [22]. Second, there must be a script that takes care of setting up GDB such that a debugging session is already set up when the user works with the interactive mode. This setup script will make use of some of the analysis methods discussed in this chapter, e. g., the import of relevant debug symbols. Finally, the webserver should be able to run on a separate machine than the analysis server. As the interactive mode needs access to parts of the core dump analysis, there must be a connection between the two machines. Therefore, on each request for an interactive session, GoTTY will connect to the analysis machine, start relevant steps of the analysis and execute the script that sets up GDB. The session will remain open as long as the user stays on the web page. The deployment of the interactive mode is discussed in Section 5.4

3.9 Core Dumps with 32-Bit Memory Alignment

Core dumps that were generated from a 32-bit operating system require a 32-bit analyzing system. At the moment, .NET core CLR does not support 32-bit Linux-based operating systems. As a consequence, the core dump analysis tool can only analyze 64-bit Linux core dumps because there is no CLR to execute the tool on a 32-bit operating system. In the Dynatrace environment, this shortcoming can be tolerated as the great majority of core dumps are created in 64-bit operating systems.

Nevertheless, the work of a .NET core CLR version that supports 32-bit Linux operating systems is in progress. The progress can be tracked via the .NET core CLR Github page: `https://github.com/dotnet/coreclr/issues/9265`. Upon release, the analysis methods should be adoptable for 32-bit core dumps with no or only little effort.

4 Integration into Development Life Cycle

Companies that develop software products for customers usually provide some way to report product misbehavior, i. e., bugs. For example, the Dynatrace company uses the issue tracking system *JIRA*. Using JIRA, customers can create tickets by providing information that is relevant for the bug. This information can consist not only of text but can also contain raw files such as Linux core dumps.

This chapter describes how the tool is integrated into the development life cycle in the Dynatrace company. The focus lies on how the analysis tool can be used in combination with JIRA such that incoming core dumps are automatically forwarded to the analysis service.

4.1 Automatic Analysis of Incoming Core Dumps

When customers issue tickets via the JIRA system, the tool is able to automatically detect and analyze Linux core dumps. After the analysis has been started, the tool creates a comment on the ticket containing a link to the analysis results. In practice almost all tickets already have this comment before a developer can have a look at it. Therefore, developers do not have to manually provide the analysis tool with the core dump but can use the analysis results immediately when viewing a ticket.

Also, it may be possible for developers to overlook dump files that are in a nested archive. The tool, however, can recursively extract archives and will always detect a Linux core dump unless the file extensions have been altered by the user.

As polling on JIRA generates a considerate amount of load on the issue tracking system, the automatic retrieval of core dumps was integrated into an already established service that checks for new tickets. In the scope of the thesis, this service has been extended to process tickets containing Linux core dumps.

The JIRA integration relies on the *JIRA Rest Java Client* (JRJC). JRJC is a Java API to access the REST interface of JIRA. The analysis tool polls in a predefined interval for new JIRA tickets. When a new ticket is found, all ticket attachments are iterated and checked whether they are Linux core dumps or archives that contain a Linux core dump. If a core dump is found, the attachment file is forwarded to the analysis process and a link to the analysis report is posted in a comment. Optionally, the comment visibility can be restricted to a certain group because it may not always be desired to make the analysis report available to the customer.

4.2 Linking Tickets with Analysis Reports

In many cases, developers want to discuss a Linux core dump with others to get help in finding out the error source or in fixing the underlying problem. Using conventional methods, all developers attending in such a discussion would need to analyze the core dump using an in-depth analysis tool. However, this proves quite time consuming and neglects people who are unexperienced in core dump analysis. By linking tickets with analysis reports, a common base for discussion is created because all developers refer to the same report. Also, the preparation time for each developer can be severely decreased.

5 Deployment

As the analysis tool was integrated into the already established SuperDump tool, some of the deployment process was already established. Although the SuperDump deployment did not need to change, additional resources were required for the Linux core dump analysis. The following sections describe the deployment that was established with SuperDump, i. e., the starting point, and how the Linux core dump analysis tool was integrated into this environment.

5.1 SuperDump Deployment

SuperDump was written in C#. Some modules rely on the .NET core framework, whereas other modules require the full .NET framework, e. g., the Windows crash dump analysis process. Therefore, the choice of the operating system came down to a Windows Server system. The server is virtualized and is running with a *VMWare* hypervisor. Figure 6 shows the original deployment of SuperDump.

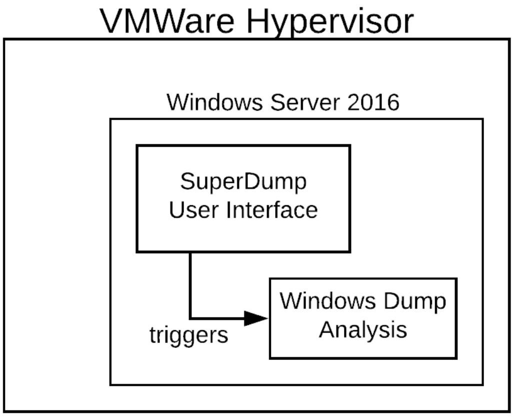


Figure 6: SuperDump Deployment. SuperDump is running on a virtualized Windows Server 2016. The Windows crash dump analysis process is running on the same machine.

5.2 Linux Analysis via Docker

The usage of *Docker* for the Linux core dump analysis provides several advantages:

- Manually setting up a machine that can run the Linux core dump analysis requires a considerate amount of effort. Section 5.3 provides details on how the image is built. Docker can automate the setup process.

- Operating system side effects can have an impact on the analysis outcome. By using Docker, most of these side effects can be prevented.

- A Docker container based on a Linux operating system can be executed on both, Windows and Linux hosts.

- By providing a Docker image, the process is scalable. It is possible to run multiple containers of the analysis image.

- Sharing the core dump analysis with Docker is simple. The image is available on Docker Hub [23] and can be downloaded and started with a single command by anyone with an installed Docker client and server.

As mentioned in Section 2.6, a separate machine for analyzing Linux core dumps is required. We chose the Linux distribution Ubuntu with version 14.04 as the operating system. Although any operating system that supports Linux containers would suffice, a Linux-based operating system can use the resources more efficiently.

Using a separate machine for analyzing Linux core dumps comes with several drawbacks. For this context, the term *host machine* refers to the machine that is running SuperDump, while the term *analyzing machine* refers to the machine running the Linux core dump analysis. The drawbacks will be explained and discussed in the following paragraphs.

- The host machine must be able to trigger an analyzation process.

- The host machine needs a way to send the Linux core dump to the analyzing machine.

- The analyzing machine needs a way to send the results back to the host machine.

- There is a performance loss due to communication between the two machines.

Remote triggering the analysis Luckily, Docker implements a feature for calling Docker commands on a remote machine. All that needs to be done is setting an environment variable with the name “DOCKER_HOST” to the IP of the target machine and subsequent Docker commands will automatically be sent to this machine.

File sharing The two machines still need a file share because on the one hand, the host machine needs to send the Linux core dump to the analyzing machine, while on the other hand, the analyzing machine needs to send the results back. Although it would be possible to transfer the analysis results to the host machine via the process standard output channel, a separate file share is the preferred choice because it provides more reliability and also allows the transmission of binary files. For file sharing, we chose the Common Internet File System (CIFS) because it is supported by default on Windows and can easily be installed on any Linux-based operating system with little effort. The host machine provides read and write access on the dump directory to the analyzing machine.

Performance loss due to communication The major loss in performance resides from transferring the Linux core dump to the analyzing machine. Naturally, this loss depends heavily on the size of the dump and can hardly be reduced. However, as the data evaluation will show, the average actual analzation time for analyzing a Linux core dump lies at about 2 minutes, whereas the time for transferring the dump takes less than a second. Therefore, in the Dynatrace environment, the average total analysis time is increased by less than one percent. The exact slowdown factor will depend on the existing environment, i. e., the performance of the analyzing machine, the network speed, and the contents of the dump. Figure 7 depicts a time diagram that shows the difference between the actual analyzation time and the total analyzation time. The actual analyzation time refers to the time solely spent on analyzing the core dump, while the total analyzation time also takes the file transfer times into account.

5.3 Analysis Images

The Linux core dump analysis tool comes with three images. The *base image* contains the operating system with all required packages installed, while the *main image* is based upon the base image and adds the Linux core dump analysis executable. The *gotty image* launches a remote debugging session over SSH with GDB. The following subsections describe the three images in more detail.

5.3.1 Linux Base Image

The Linux base image provides an environment that can be used by other images. It comes with several preinstalled tools, such as GDB, .NET core CLR, and OpenSSH. The complete list of installed packages including the reasoning is described in Table 8. The image copies libunwind sources from the local filesystem and compiles them. This way, changes to the libunwind source can be applied to the existing environment by recreating the base image. Listing 8 shows the contents of the according Dockerfile which contains four layers. The first layer represents the operating system, namely the most recent version of Ubuntu. The second layer is generated by copying libunwind sources to */libunwind*, while the third and fourth layer are created using concatenated

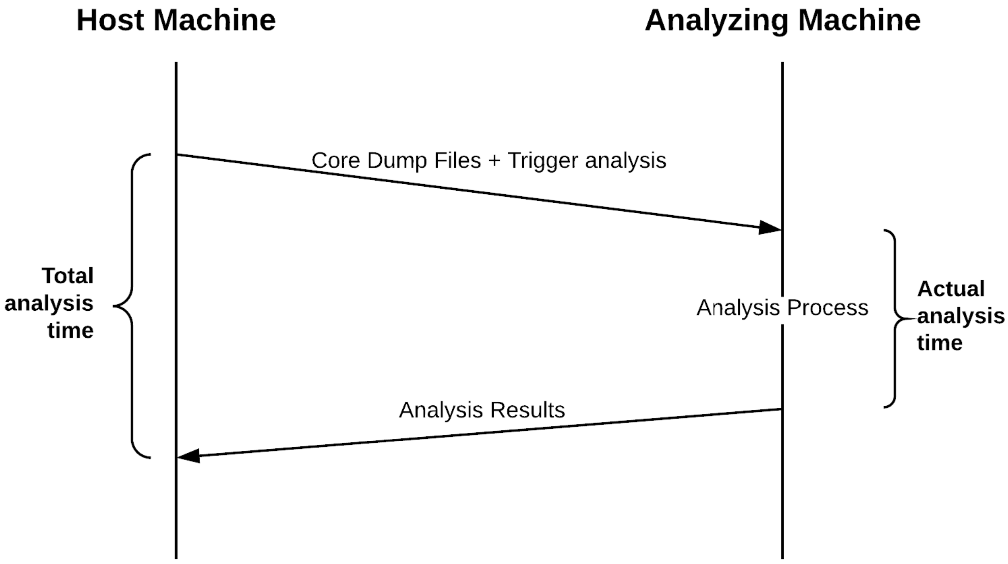


Figure 7: Time Diagram of a Core Dump Analysis

commands. There is no *CMD* instruction because this image is only meant to be used as a base image for other images.

The split into a base and a main image provides developers with faster building times because the base image does not need rebuilding unless new packages are introduced. If the core dump analysis tool changes, the main image can be rebuilt without rebuilding the base image which saves a considerate amount of time.

Package	**Reason**
build-essential	This package is required to compile and install libunwind.
GDB	GDB is used by the analysis tool as well as by the interactive mode.
.NET core CLR	The analysis tool is running with the .NET core CLR.
rsync	Rsync is used to create a working copy of the core dump for either the analysis or the interactive mode.
OpenSSH Server	When starting the interactive mode, a remote SSH session to the machine hosting the core dumps is initialized.

Table 8: The Linux base image installs several packages that are required by the core dump analysis tool.

```
FROM ubuntu

COPY . /libunwind

RUN apt-get update && \
  apt-get install -y apt-transport-https && \
  echo "deb␣[arch=amd64]␣https://apt-mo.trafficmanager.net/repos/dotnet-release/␣
      xenial␣main" > /etc/apt/sources.list.d/dotnetdev.list && \
  apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 417A0893 && \
  apt-get update && \
  apt-get upgrade -y && \
  apt-get install -y build-essential gdb dotnet-dev-1.0.4 rsync openssh-server &&
      \
  apt-get clean

RUN cd /libunwind && \
  ./configure CFLAGS="-fPIC" && \
  make && \
  make install && \
  rm -R /libunwind
```

Listing 8: Base Dockerfile. The Dockerfile for the Linux base image sets up the environment for the core dump analysis tool. Besides installing required packages, it also installs libunwind by compiling its sources.

Please note the structure of the *RUN* commands. By concatenating commands with the && notation, only a single layer is created although multiple commands are executed. The reason for keeping the layer count small is to reduce the disk space consumption. Concatenating correlating commands assures that layers do not contain unnecessary intermediate files that are not used in the final image.

The Linux base image can be built by using the provided Dockerfile and the command *docker build -f Dockerfile.Linux.Base -t dotstone/sdlinux:base* or by downloading the image from `https://hub.docker.com` with the Docker command *docker pull dotstone/sdlinux:base*. However, the latter approach is recommended because it assures that the image with the most recent version is downloaded.

5.3.2 Main Image

The main image builds upon the Linux base image and adds the core dump analysis tool. Its Dockerfile is shown in Listing 9. First, Docker imports the binaries of the core dump analysis tool as well as the source code of the libunwind wrapper. Then, the libunwind wrapper gets compiled through a series of compiler commands. Finally, the default command for execution is set to trigger the core dump analysis process. This command consists of several subcommands because it needs to copy the dump to a temporary directory that can be manipulated without any side effects. After the analysis is finished, all log files that were produced are copied to the original dump directory.

```
FROM dotstone/sdlinux:base

COPY LibunwindWrapper /wrappersrc
COPY SuperDump.Analyzer.Linux /opt/SuperDump.Analyzer.Linux

RUN /bin/bash -c "g++␣-c␣-x␣c++␣/wrappersrc/LibunwindWrapper.cpp␣-I␣/usr/local/
    include␣-o␣"/opt/LibunwindWrapper.o"␣-fpic␣-std=c++11;␣g++␣-c␣-x␣c++␣/
    wrappersrc/Main.cpp␣-I␣/usr/local/include␣-o␣"/opt/Main.o"␣-fpic␣-std=c++11;␣g
    ++␣-o␣"/usr/lib/unwindwrapper.so"␣-Wl,-L"/usr/lib/x86_64-linux-gnu"␣-Wl,-L"/
    lib/x86_64-linux-gnu"␣-shared␣/opt/LibunwindWrapper.o␣/opt/Main.o␣/usr/local/
    lib/libunwind-coredump.a␣-l"unwind-x86_64";mkdir␣/opt/dump"

CMD rsync -a /dump /opt/ && cd /opt/dump/ && dotnet /opt/SuperDump.Analyzer.Linux/
    SuperDump.Analyzer.Linux.dll /opt/dump/ /dump/superdump-result.json && cp /opt
    /dump/*.log /dump/
```

Listing 9: Linux Analysis Dockerfile

Using a copy of the core dump yields several advantages. As discussed in Section 3.3, disk space requirements are decreased because the debug files do not have to be merged into the binary files. When using a temporary copy, each debug symbol file only needs to be stored once. Furthermore, we can guarantee that the core dumps do not change by merely analyzing them, e. g., through a bug in the analysis. Another benefit is that files on the file share are not directly accessed by the analysis tool. A direct access could lead to incomplete analyses, e. g., if the network is interrupted while analyzing. Instead, by copying the core dump first, either the copy process fails, resulting in a failed analysis, or the analysis will be successful. In general, this is the preferred solution because analysis errors are detected immediately.

5.3.3 GoTTY Image

While the previous images are targeted to perform the analysis, the purpose of the GoTTY image is to provide a user interface for the interactive mode. The deployment of the interactive mode will be described in detail in Section 5.4.

```
FROM ubuntu
RUN apt-get update; apt-get install -y wget openssh-client sshpass
RUN wget -O /root/gotty.tar.gz https://github.com/yudai/gotty/releases/download/v1
    .0.0/gotty_linux_amd64.tar.gz; gunzip /root/gotty.tar.gz; tar xf /root/gotty.
    tar -C /root/; echo SendEnv REPOSITORY_URL >> /etc/ssh/ssh_config; echo
    SendEnv REPOSITORY_AUTH >> /etc/ssh/ssh_config
CMD /root/gotty -p "3000" --title-format "GDB␣Interactive␣Session␣({{␣.Hostname␣
    }})" -w --permit-arguments sshpass -p $SSH_PASSWD ssh -t -o
    StrictHostKeyChecking=no $SSH_USER@$SSH_HOST /opt/SuperDump.Analyzer.Linux/gdb
    .sh
```

Listing 10: GoTTY Dockerfile

The Dockerfile shown in Listing 10 takes care of setting up an environment for GoTTY such that users will be automatically connected to the analysis machine via SSH with a GDB session already set up. In particular, Docker will install required packages: *wget* for downloading GoTTY, *OpenSSH-Client* for connecting to the analysis machine, and *sshpass* to allow specifying the SSH password as an argument. Please note that using an argument to specify a password is highly unsecure. In this context, however, security is not at risk because the analysis machine is properly secured against non root users. In fact, the password that is used for debugging sessions is considered as publicly available and according security measures are in place. The last step of the setup installs version 1.0.0 of GoTTY and configures SSH to forward the two environment variables *REPOSITORY_URL* and *REPOSITORY_AUTH* to the target machine. These two environment variables are required to get access to the source repository. The default execution command starts a GoTTY instance on port 3000 that connects to the analysis machine for each incoming request. Placeholders starting with a dollar sign are set in a separate configuration file. This configuration file as well as the contents of the GDB setup script will be described in Section 5.4.

5.4 Interactive Mode via Docker Compose

As Docker was used for the core dump analysis, Docker Compose was the first choice for deploying the interactive mode. Nevertheless, Docker Compose comes with two major advantages that set it apart from manual deployment approaches.

- Configuration: Maintenance effort is greatly decreased because Docker Compose centralizes and standardizes the configuration of multiple services that may run on multiple physical machines the same way Docker does with single containers.

- Scalability: With Docker Compose, individual services can be scaled independently and dynamically at run-time. For example, if the webserver for the interactive mode is reacting slow due to a high load, more webserver instances can be launched by invoking the Compose command *docker-compose scale gotty=X* where X stands for the desired number of instances. Of course, the same goes for the service running GDB. Downscaling can be done with the same command but providing a lower instance count. Please note that Compose merely adds or removes instances to match the specified scale count in order to reduce service downtime to a minimum. Scaling is done with the help of a load balancer that is running on each machine.

Therefore, Docker Compose fits with all requirements that were described in Section 3.8.1, while keeping maintenance effort for deployment low. The Compose file for the interactive mode is shown in Listing 11. Two services are defined: the *GoTTY* service and the *interactive* service. The *GoTTY* service resembles the webserver that is running an instance of GoTTY and the *interactive* service resembles the debugger. Please note that the image used in the interactive service is the image used for core dump analysis. Therefore, no additional image is needed and the analysis command can be used without any further setup. The GoTTY service defines a few environment variables for the SSH connection and the GoTTY configuration. Also, port 3000 is forwarded to provide access to the web user interface. The interactive service sets up a running SSH server and mounts the dump directory as well as the directory that holds the debug symbols. These two directories are mounted in read-only mode because the interactive mode does not alter any files. Instead, the interactive mode will rely on debug symbols that were retrieved when the core dump was analyzed.

```
version: ’3.1’
services:
    gotty:
        image: dotstone/gotty
        ports:
            - "3000:3000"
        environment:
            GOTTY_PORT: "3000"
            GOTTY_ADDRESS: "127.0.0.1"
            SSH_HOST: "interactive"
            SSH_USER: "gdb"
            SSH_PASSWD: "gdb"
            REPOSITORY_URL: "${REPOSITORY_URL}"
            REPOSITORY_AUTH: "${REPOSITORY_AUTH}"
    interactive:
        image: dotstone/sdlinux
        command: bash -c "useradd␣-s␣/bin/bash␣gdb;␣echo␣gdb:gdb␣|␣chpasswd;␣chown
            ␣-R␣gdb:gdb␣/opt/dump;␣mkdir␣/var/run/sshd;␣echo␣AcceptEnv␣
            REPOSITORY_URL␣>>␣/etc/ssh/sshd_config;␣echo␣AcceptEnv␣REPOSITORY_AUTH
            ␣>>␣/etc/ssh/sshd_config;␣/usr/sbin/sshd␣-D"
        volumes:
            - ${DUMP_DIR}:/dumps:ro
            - ${DEBUG_SYMBOL_DIR}:/debugsymbols:ro
        expose:
            - "22"
```

Listing 11: GoTTY Docker Compose file

5.5 Limitations

Several advantages of the deployment methods were explained in this chapter. However, the deployment approach also comes with disadvantages which shall be described in following paragraphs.

Additional Prerequisite for SuperDump Using Docker for deployment requires every user that wants to use the Linux core dump analysis service to install Docker. More so, building the core dump analysis service also requires Docker to be installed because the analysis image has to be created. Thus, the build process becomes more complex.

Windows Containers on the Host Machine Docker allows only Windows or Linux containers at a time. By requiring a Linux container for the core dump analysis, Windows containers are no longer available on the SuperDump machine even if the Linux container is running on a remote host. Although technically possible, dynamically switching between Linux and Windows container mode is highly discouraged as it would make analysis times unpredictable.

6 User Interface

The user interface that was developed within this thesis can be split into two parts: the analysis report which is the primary source of information for users and the statistical evaluation which serves for creating charts about analyzed dumps. While each analysis report refers to a single core dump, the statistics refer to a set of core dumps. The two visualization parts are described in this chapter.

6.1 Analysis Report

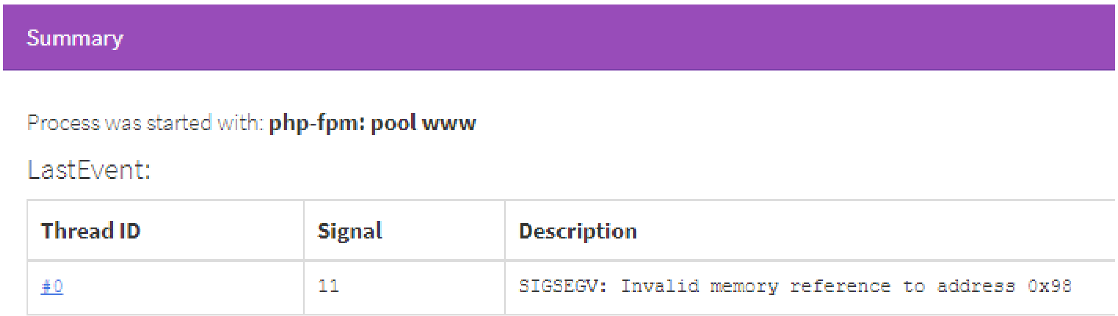
Before introducing the Linux core dump analysis, SuperDump was only able to display Windows crash dump analysis results. These results are stored in a single JSON file. In order to reduce the required effort to display core dump analysis results and keep consistency across the two types of dump results, this format has been chosen for the core dump analysis too. However, several adaptations to the visualization were required because the contents of Linux core dumps are different from the content of Windows crash dumps. For example, Windows stores information about the last event that happened before a crash in the crash dump, whereas Linux stores signal information.

Another advantage of sticking with the Windows crash dump visualization is that many users of the Linux core dump analysis were already used to this visualization. In fact, most users that are using the core dump analysis have been working with SuperDump before. Therefore, a certain consistency between Windows crash dump and Linux core dump visualization was demanded.

The analysis report is split into four parts: summary, system information, thread report, and module information. These parts are described in the following subsections. Each visualization part is explained with a screenshot that shows the according analysis output. The underlying core dump was found in a real world environment and stays consistent throughout the section.

6.1.1 Summary

The summary section provides information about how the process was started and the signal that caused the termination. Figure 8 shows an example output of the summary section. First, the exact call that was used for starting the process is stated. The table below explains the signal that was retrieved prior to the crash. This table contains the thread that retrieved the signal, the signal number, and a signal explanation that may provide additional information to the crash depending on the type of signal. For example, for a segmentation fault with the signal number 11, the description contains the memory address that was tried access.



Summary

Process was started with: **php-fpm: pool www**

LastEvent:

Thread ID	Signal	Description
#0	11	SIGSEGV: Invalid memory reference to address 0x98

Figure 8: Summary tab. The summary tab indicates that the crashing process was started with the call: *php-fpm: pool www*. The table on the bottom explains the signal that caused the crash. The process crashed due to a segmentation fault while trying to access the memory at address 0x98.

Field	**Description**
Platform	Short description of the platform that the process was running on
User ID	User ID of the user that was used for executing the process
Group ID	Group ID of the executing user
Entry Point	The entry point is a pointer to the first instruction that will be executed
Modules	Shows how many modules were loaded. Additionally, tagged modules will be listed below
Threads	Shows how many threads were running at the time of the crash. Additionally, tagged threads will be listed below

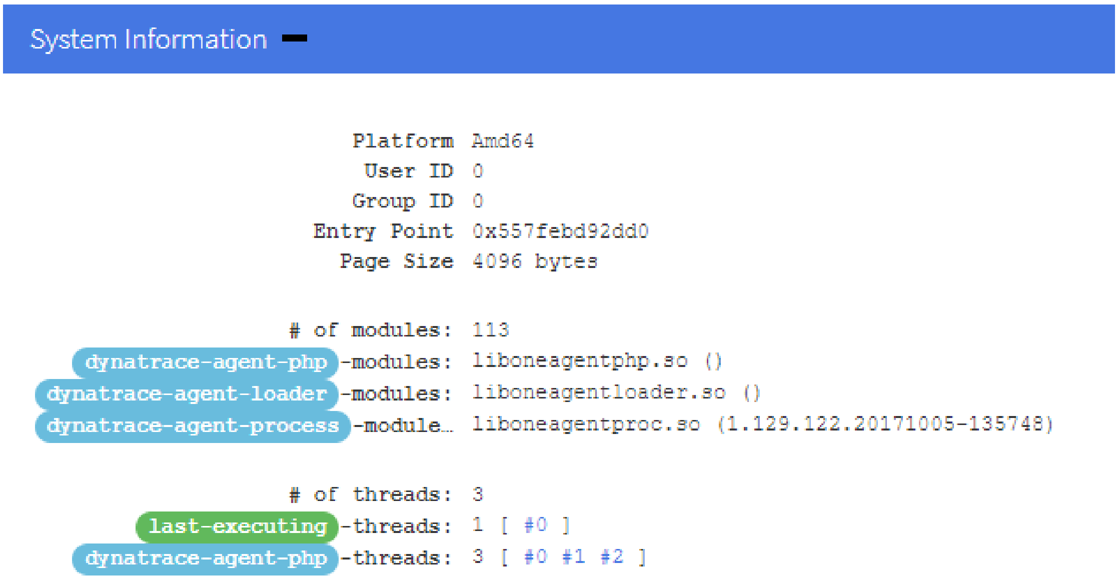
Table 9: Fields that are displayed in the system information tab of the analysis report

6.1.2 System Information

The system information tab provides a brief overview of the system and the executed process. The contents are divided into three sections: the first section shows information about the platform and the executable file, the second section shows details about shared libraries, and the third section shows details about threads that were running when the process crashed. Table 9 lists the fields that are displayed in the system information tab, while Figure 9 depicts the system information output of an example analysis report.

6.1.3 Thread Report

The thread report lists all threads that were running when the process terminated. Each thread is rendered as an expandable tab to provide more information on demand without cluttering the user interface. A specific subset of these tabs can also be opened by using the open/close buttons on top. Further, colored thread tags are displayed in the tab header which is especially useful when searching for a concrete thread. For example, the thread that caused the crash can be figured out without investigating the details of any



System Information −

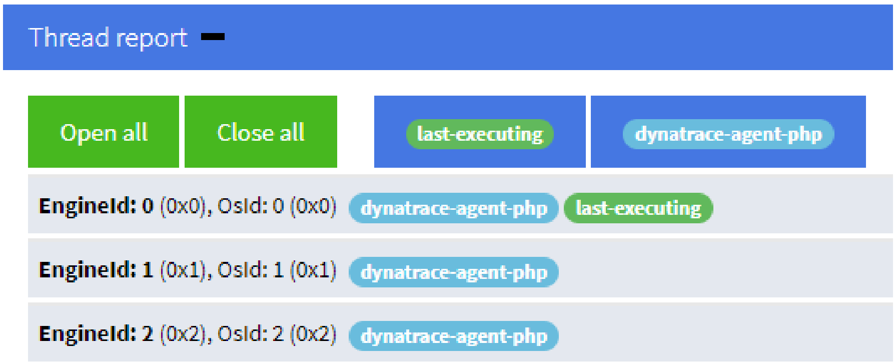
```
                         Platform  Amd64
                          User ID  0
                         Group ID  0
                      Entry Point  0x557febd92dd0
                        Page Size  4096 bytes

                    # of modules:  113
   dynatrace-agent-php-modules:  liboneagentphp.so ()
 dynatrace-agent-loader-modules:  liboneagentloader.so ()
dynatrace-agent-process-module…  liboneagentproc.so (1.129.122.20171005-135748)

                    # of threads:  3
         last-executing-threads:  1 [ #0 ]
    dynatrace-agent-php-threads:  3 [ #0 #1 #2 ]
```

Figure 9: The system information tab shows information that is specific to the executing machine and the process. This screenshot is taken from a core dump that ran on a 64 bit system and a page size of 4096 bytes. The process loaded 113 modules dynamically and had three threads when it crashed. Some of the modules and threads are tagged with Dynatrace specific tags. The user and group id of zero indicates that the process was run with root privileges.

thread because there is only one thread with the *last-executing* tag.

Figure 10 shows a screenshot of a sample thread report in the initial view, while Figure 11 shows the expanded view for the first thread. The expanded view shows the thread ID which is once denoted as *Engine thread ID* and once as *OS thread ID* because the same visualization is shared between Linux and Windows dump results and Windows processes have both IDs.

The stacktrace is shown in a table below. Among various addresses such as the instruction pointer and the return address, the most important content of the stacktrace is provided in the column *Module/Method name*. The module name represents the file name of the library that the instruction pointer belongs to. This module can also be retrieved by checking the instruction pointer against the memory mapping in the modules tab which will described in the next subsection. The method name corresponds to the demangled method name and includes the parameter list. The last part of the information shown in this column is the offset from the beginning of the memory mapping of the according module. The module is separated from the method name with an exclamation mark, while the method name is divided from the offset with a plus sign.



Thread report

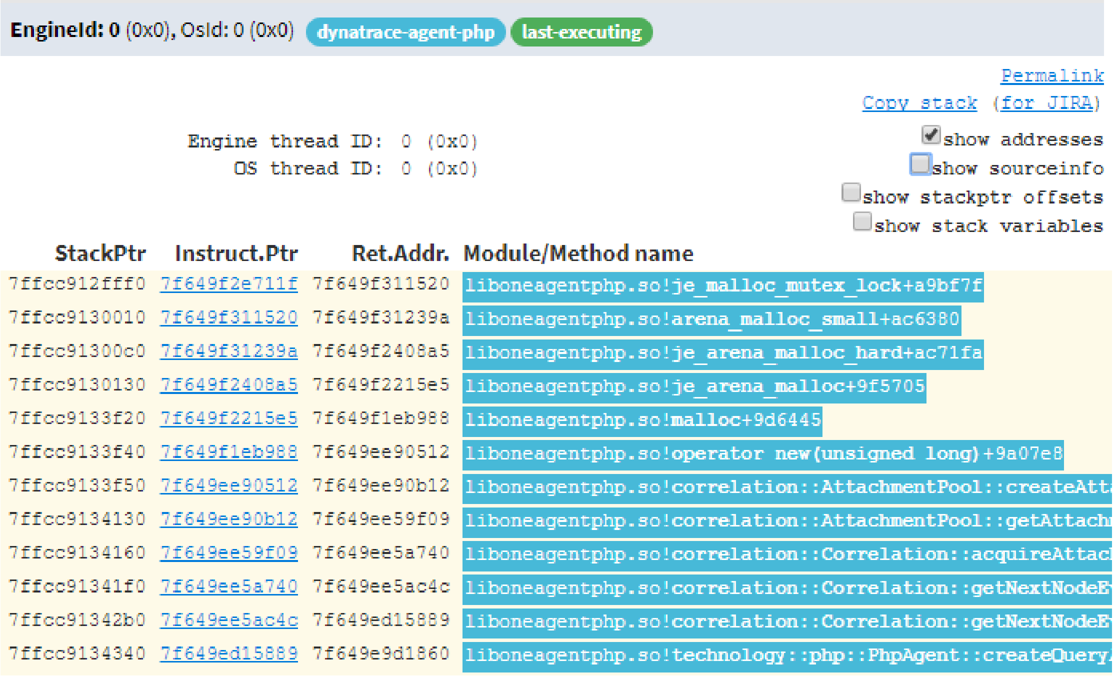
Open all | Close all | last-executing | dynatrace-agent-php

EngineId: 0 (0x0), OsId: 0 (0x0) dynatrace-agent-php last-executing

EngineId: 1 (0x1), OsId: 1 (0x1) dynatrace-agent-php

EngineId: 2 (0x2), OsId: 2 (0x2) dynatrace-agent-php

Figure 10: Thread report. The thread report contains a list of threads that were executed. Initially, the thread tabs are collapsed but can be expanded with a click on the according tab. The *Open all* and the *Close all* buttons as well as the tag buttons can be used for collapsing and expanding a number of threads more conveniently. The thread report shown has three threads running which were all tagged with the *dynatrace-agent-php* tag. The first thread carries an additional *last-executing* tag.



EngineId: 0 (0x0), OsId: 0 (0x0) dynatrace-agent-php last-executing

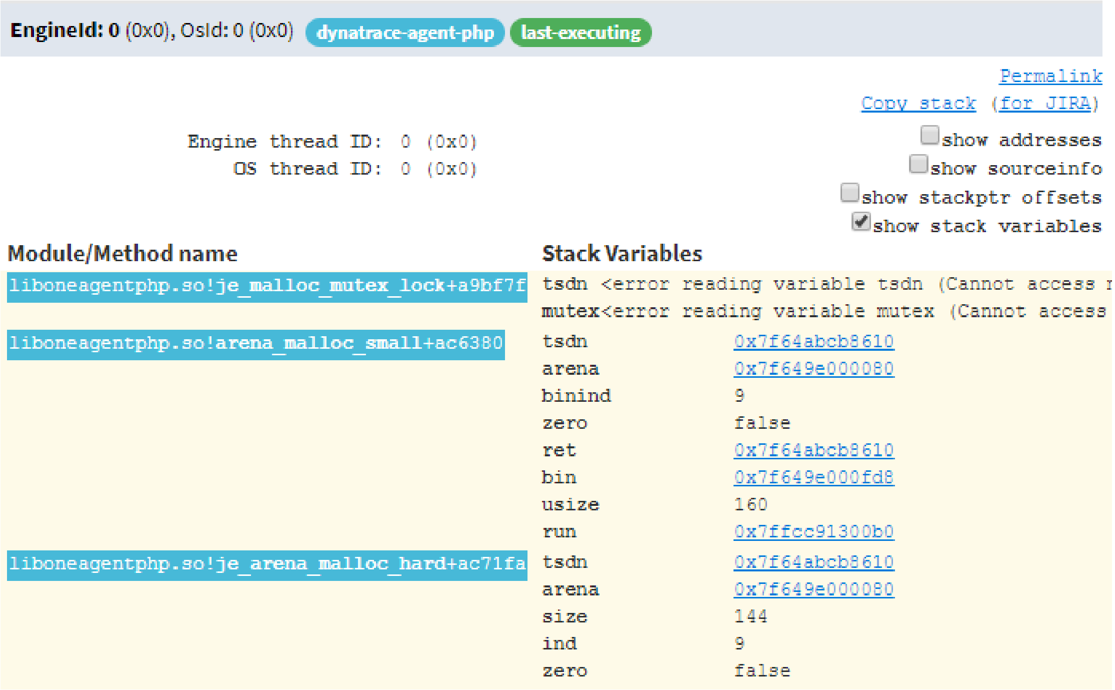
Permalink
Copy stack (for JIRA)

```
Engine thread ID:  0  (0x0)
    OS thread ID:  0  (0x0)
```

☑ show addresses
☐ show sourceinfo
☐ show stackptr offsets
☐ show stack variables

StackPtr	Instruct.Ptr	Ret.Addr.	Module/Method name
7ffcc912fff0	7f649f2e711f	7f649f311520	liboneagentphp.so!je_malloc_mutex_lock+a9bf7f
7ffcc9130010	7f649f311520	7f649f31239a	liboneagentphp.so!arena_malloc_small+ac6380
7ffcc91300c0	7f649f31239a	7f649f2408a5	liboneagentphp.so!je_arena_malloc_hard+ac71fa
7ffcc9130130	7f649f2408a5	7f649f2215e5	liboneagentphp.so!je_arena_malloc+9f5705
7ffcc9133f20	7f649f2215e5	7f649f1eb988	liboneagentphp.so!malloc+9d6445
7ffcc9133f40	7f649f1eb988	7f649ee90512	liboneagentphp.so!operator new(unsigned long)+9a07e8
7ffcc9133f50	7f649ee90512	7f649ee90b12	liboneagentphp.so!correlation::AttachmentPool::createAtt
7ffcc9134130	7f649ee90b12	7f649ee59f09	liboneagentphp.so!correlation::AttachmentPool::getAttach
7ffcc9134160	7f649ee59f09	7f649ee5a740	liboneagentphp.so!correlation::Correlation::acquireAttac
7ffcc91341f0	7f649ee5a740	7f649ee5ac4c	liboneagentphp.so!correlation::Correlation::getNextNodeE
7ffcc91342b0	7f649ee5ac4c	7f649ed15889	liboneagentphp.so!correlation::Correlation::getNextNodeE
7ffcc9134340	7f649ed15889	7f649e9d1860	liboneagentphp.so!technology::php::PhpAgent::createQuery

Figure 11: Screenshot of the Expanded Thread Report. Expanding the first thread in the thread report tab reveals the stacktrace. The first frames in the stacktrace originate from the module *liboneagentphp.so*. This could be a potential starting point for debugging. The stacktrace table was cut off in order to fit into the page.



EngineId: 0 (0x0), OsId: 0 (0x0) dynatrace-agent-php last-executing

Permalink
Copy stack (for JIRA)

```
Engine thread ID:  0  (0x0)
    OS thread ID:  0  (0x0)
```

□show addresses
□show sourceinfo
□show stackptr offsets
☑show stack variables

Module/Method name	Stack Variables	
liboneagentphp.so!je_malloc_mutex_lock+a9bf7f	tsdn <error reading variable tsdn (Cannot access r	
	mutex<error reading variable mutex (Cannot access	
liboneagentphp.so!arena_malloc_small+ac6380	tsdn	0x7f64abcb8610
	arena	0x7f649e000080
	binind	9
	zero	false
	ret	0x7f64abcb8610
	bin	0x7f649e000fd8
	usize	160
	run	0x7ffcc91300b0
liboneagentphp.so!je_arena_malloc_hard+ac71fa	tsdn	0x7f64abcb8610
	arena	0x7f649e000080
	size	144
	ind	9
	zero	false

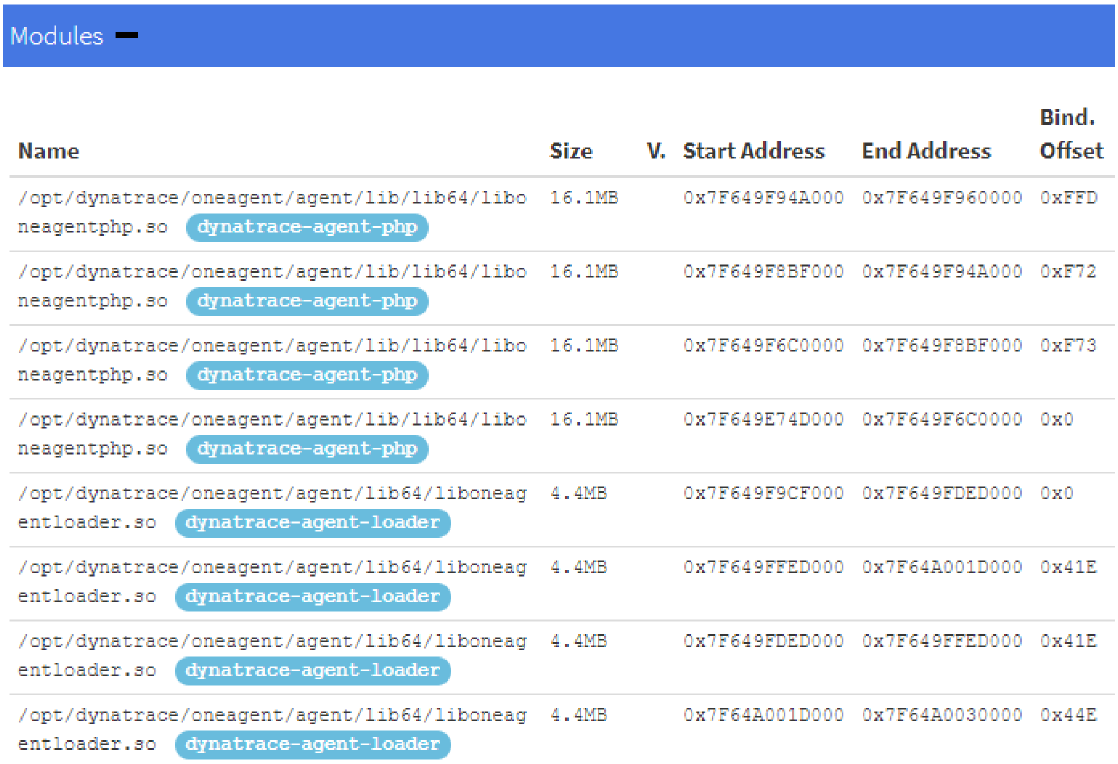
Figure 12: Stack variables are displayed in a separate column if the *show stack variables* checkbox is ticked.

On the top right of the view, there are four checkboxes that serve for showing and hiding information in the stacktrace table.

- The *show addresses* checkbox toggles the visibility of the three address fields *Stack Pointer*, *Instruction Pointer*, and *Return Address*.
- *show sourceinfo* is used to display source information after the method name, i. e., source file and line number.
- *stackptr offsets* were not implemented in the Linux analysis as stack pointer offsets usually do not provide a significant benefit for debugging. The checkbox originates from the Windows analysis where this functionality is implemented.
- *show stack variables* toggles the visibility of an additional column on the very right of the table. This column contains the names and values of stack variables, i. e., local variables and function arguments. Figure 12 shows how the output looks like. Stack variables are only available if debug symbol files for the according module are present.

6.1.4 Module Information

The last section of the analysis report shows information about shared libraries as shown in Figure 13. This information consists of:



Modules

Name	Size	V.	Start Address	End Address	Bind. Offset
/opt/dynatrace/oneagent/agent/lib/lib64/liboneagentphp.so dynatrace-agent-php	16.1MB		0x7F649F94A000	0x7F649F960000	0xFFD
/opt/dynatrace/oneagent/agent/lib/lib64/liboneagentphp.so dynatrace-agent-php	16.1MB		0x7F649F8BF000	0x7F649F94A000	0xF72
/opt/dynatrace/oneagent/agent/lib/lib64/liboneagentphp.so dynatrace-agent-php	16.1MB		0x7F649F6C0000	0x7F649F8BF000	0xF73
/opt/dynatrace/oneagent/agent/lib/lib64/liboneagentphp.so dynatrace-agent-php	16.1MB		0x7F649E74D000	0x7F649F6C0000	0x0
/opt/dynatrace/oneagent/agent/lib64/liboneagentloader.so dynatrace-agent-loader	4.4MB		0x7F649F9CF000	0x7F649FDED000	0x0
/opt/dynatrace/oneagent/agent/lib64/liboneagentloader.so dynatrace-agent-loader	4.4MB		0x7F649FFED000	0x7F64A001D000	0x41E
/opt/dynatrace/oneagent/agent/lib64/liboneagentloader.so dynatrace-agent-loader	4.4MB		0x7F649FDED000	0x7F649FFED000	0x41E
/opt/dynatrace/oneagent/agent/lib64/liboneagentloader.so dynatrace-agent-loader	4.4MB		0x7F64A001D000	0x7F64A0030000	0x44E

Figure 13: Screenshot of the module tab

- Module path: The full path to the shared library that was used when the process was executing.
- Tags: If available, tags are displayed after the module path.
- Size: File size of the shared library file.
- Version: The version of the shared library.
- Start/end address, binding offset: The mapping addresses that were used when recreating the memory image. While there can be multiple entries with the same shared library, there must never be overlaps between the mapping addresses.

6.2 Statistics

This section deals with the statistical evaluation concerning dump analysis data. The main purpose of providing statistics is to answer questions such as:

- How often did a specific application crash in the last couple of days?

- Was there a peak in crashes over a specific period of time? And if so, which application(s) caused the peak?
- Are there any shared libraries that tend to crash more often than others?

The Sections 6.2.1 and 6.2.2 explain how Elasticsearch and Kibana were set up, while Subsection 6.2.3 provides some sample analyses that provide answers to the three questions mentioned. Subsection 6.2.4 describes the limitations of the statistical evaluation.

6.2.1 Elasticsearch Index Fields

The Elasticsearch index does not require all dump report data. Some fields, e. g., stacktraces, would not provide any use for the analyzation and were therefore omitted in the Elasticsearch index. Other fields were duplicated to enable data queries without doing any computations, e. g., the module information is stored once for all modules, once for modules with version information, and once for Dynatrace modules with version information. This kind of redundancy is common practice in Elasticsearch indices. The fields that are stored in the Elasticsearch index are shown in Table 10. Please note, that there is only one index for both, Windows and Linux dump reports.

The text files are used as the primary data storage because the Elasticsearch fields can be derived from the text files but not the other way around. The Elasticsearch index is consistently synchronized with the text files.

6.2.2 Visualization

Elasticsearch can store and analyze the provided data but separate tools are required for visualization. The two most common tools are Kibana [24] and Grafana [25]. Both tools provide a powerful and intuitive way of visualizing data stored in Elasticsearch. While the focus of Grafana lies on specific metrics, Kibana focuses on log analytics [26]. However, both tools are perfectly capable of satisfying the use cases for visualizing dump report data. In the end, Kibana was chosen over Grafana because Kibana has a better support for aggregations, i. e., combining two indices in a single visualiation. This benefit might be useful in the future if dump report data cannot be stored in a single index but must be stored over multiple indices.

6.2.3 Example Visualizations

This Section shows how to answer some common questions about dump data. The screenshots provided were taken from the live environment. The data that was analyzed contains both, Windows and Linux dump reports. Kibana provides a very intuitive way for creating visualizations:

1. Open the *Visualize* tab on the left
2. Click on the + icon to create a new visualization

Name	**Description**
id	The ID is a unique identifier of the core dump
analyzationDurationSecs	The duration of the analysis in seconds
dumpSize	The size of the plain core dump in bytes
dynatraceLoadedModulesVersioned	The list of dynatrace modules with version information
executable	The path of the executable
exitType	The signal code that lead to the termination
isManaged	Indicates if the process was a managed process. This flag is always false for Linux core dumps.
lastEventDescription	A textual description of the last event before the termination. For Linux core dumps, a simple description of the signal code is inserted, while Windows stores a separate crash description in each dump file.
loadedModules	All modules without version information
loadedModulesVersioned	All modules with version information
nrThreads	The number of running threads at the time of the crash
processArch	Processor architecture
systemArch	System architecture. Usually the system architecture equals the process architecture but they might differ in particular cases.
timestamp	The date and time when the core dump was analyzed
type	This flag is used to distinguish between Windows crash dumps and Linux core dumps

Table 10: Fields that are stored in the Elasticsearch index including a description

3. Select the visualization type, e. g., a vertical bar chart

4. Select the index that contains the data

5. The right pane shows the current visualization. On the left pane, you can modify the axis and various properties of the visualization.

6. Optionally, you can save the visualization by clicking *Save* on the top right corner and providing a name

Question 1: How often did a specific application crash in the last couple of days?

If a given application is suspect to crashing frequently, the answer to this question can yield interesting results. First, we create a new visualization using a vertical bar chart. We can then set the timeframe to the last 7 or the last 14 days. The chart displays the total number of core dumps that were analyzed in the given period. In order to restrict the dumps to a specific application, we click on the *Split Chart* option in the buckets panel and select *Filters* in the aggregation field. A textbox pops up that allows us to specify a query in the *Lucene Query String* syntax [27]. In this example, we want to get dumps that originated from an application called *php-fpm*. However, we need to specify the full path of the executable, i. e., "./usr/sbin/php-fpm". The final query is: "executable:./usr/sbin/php-fpm". When updating the chart, the number of php-fpm crashes is displayed which answers our question.

Nevertheless, it would be interesting to see how these crashes transitioned over time. We can add another sub-bucket by clicking on the according button in the buckets panel. This time, we select *X-Axis* and set the aggregation type to *Date Histogram*. Optionally, we can specify an interval if the automatically detected interval does not fit our needs. The final chart is shown in Figure 14.

Question 2: Was there a peak in crashes over a specific period of time? And if so, which application(s) caused the peak?

The data retrieved from crash dump analysis can be used to detect crash anomalies. A peak in crashes may indicate a critical bug in the software.

For this type of question, we can use a vertical bar chart again. After setting the timeframe up accordingly, we need to add an X-Axis and set the aggregation type to *Date Histogram*. We can already check the chart to see if there are any peaks. Figure 15 shows the current chart for a timeframe of about three weeks. We can see that there was a crash peak in the end of October 2017.

In order to see which applications were crashing at the time in question, we can click on the according bar. A filter is created that limits the crashes to the date selected. With this filter active, we can set the aggregation type of the X-Axis to *Term* because we do not need the *Date Histogram* anymore. The *executable* field must be selected in the term textbox. Optionally, we can modify the number of displayed items with the size input field. The resulting chart shows which applications were crashing the most in

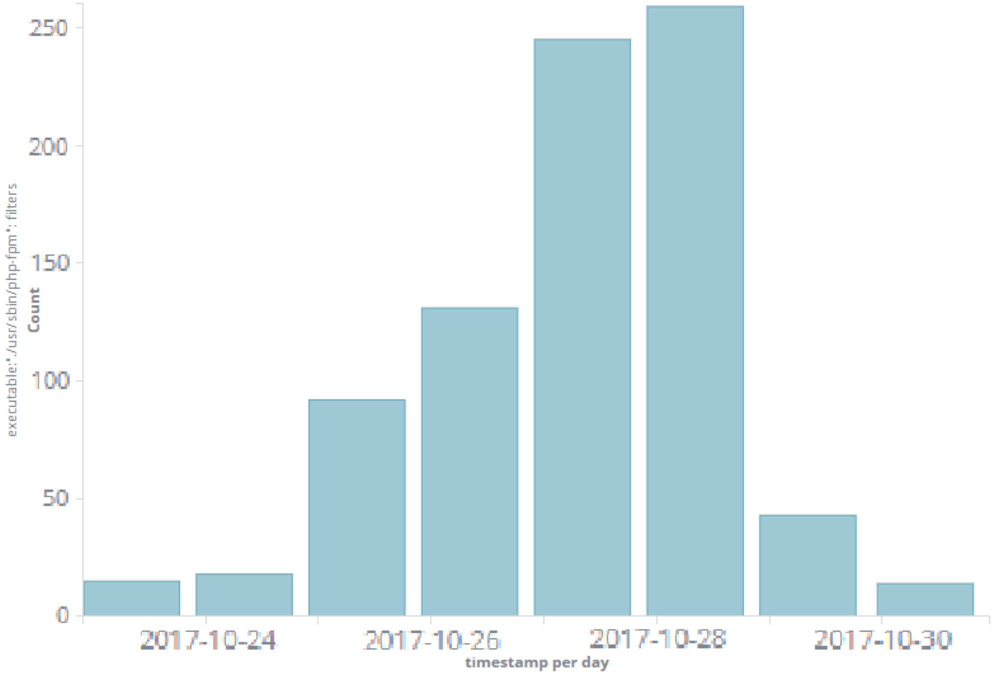


Figure 14: Example Visualization 1. How often did php-fpm crash in the last 7 days?

the selected timeframe as depicted in Figure 16. We can see that *php-fpm* was causing the crash peak with almost 250 crashes in a single day.

Question 3: Are there any shared libraries that tend to crash more often than others?

Not only the executable can be the problem source for a crash but also shared libraries. This question is similar to question 2. Again, we start by creating a vertical bar chart and setting the timeframe accordingly. This time, we set the X-Axis aggregation to *Terms* and select *loadedModules* from the dropdown menu. Unfortunately, the chart does not provide useful insights on the crash-likeliness of specific modules because all bars have similar counts. The reason for this is that the most frequent modules are loaded in all applications and are thus listed in each core dump. In order to get a valid statement on the crash-likeliness, we would need to know about the applications that were not crashing.

Still, we can use the data to check the total number of crashes for specific modules. By limiting the list of modules to only critical ones, we get a chart like the one shown in Figure 17. Instead of using *loadedModules*, the field *dynatraceLoadedModulesVersioned* was used. This field limits the modules to only those manufactured by Dynatrace and also contains version information. The chart shows that the module *dtagent.dll* with version 6.5.0.1289 was crashing the most, followed by *dtagentcore.dll* with the same version.

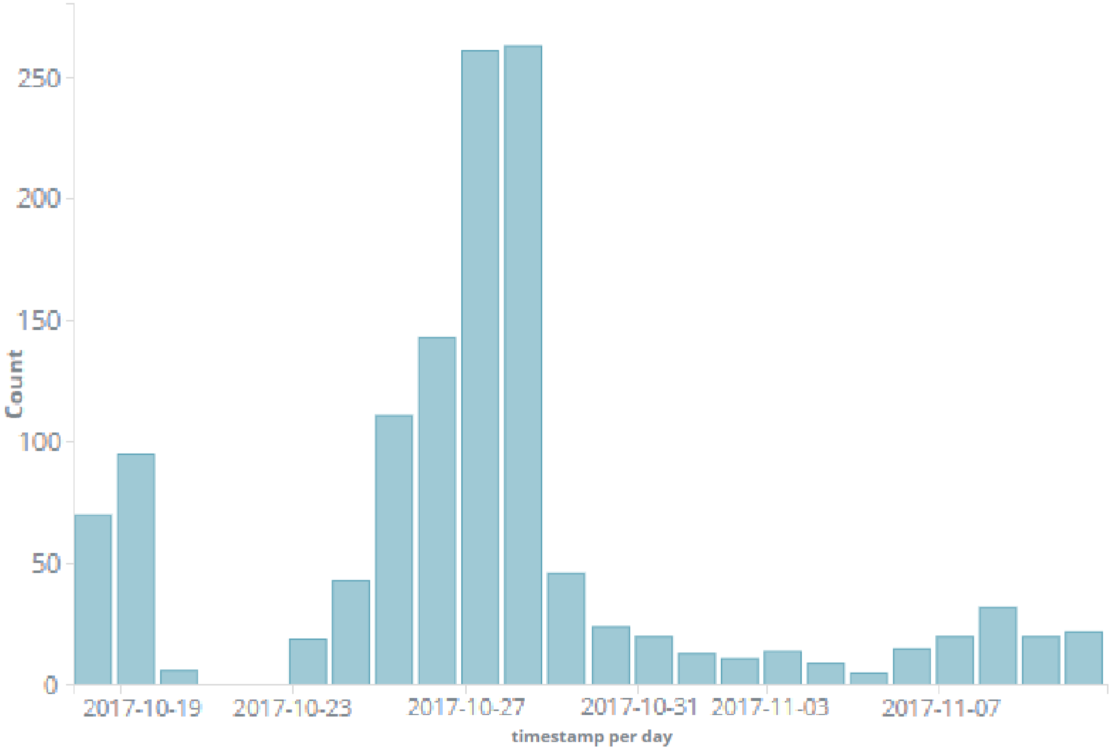


Figure 15: Example Visualization 2. Was there a peak in crashes over a specific period of time?

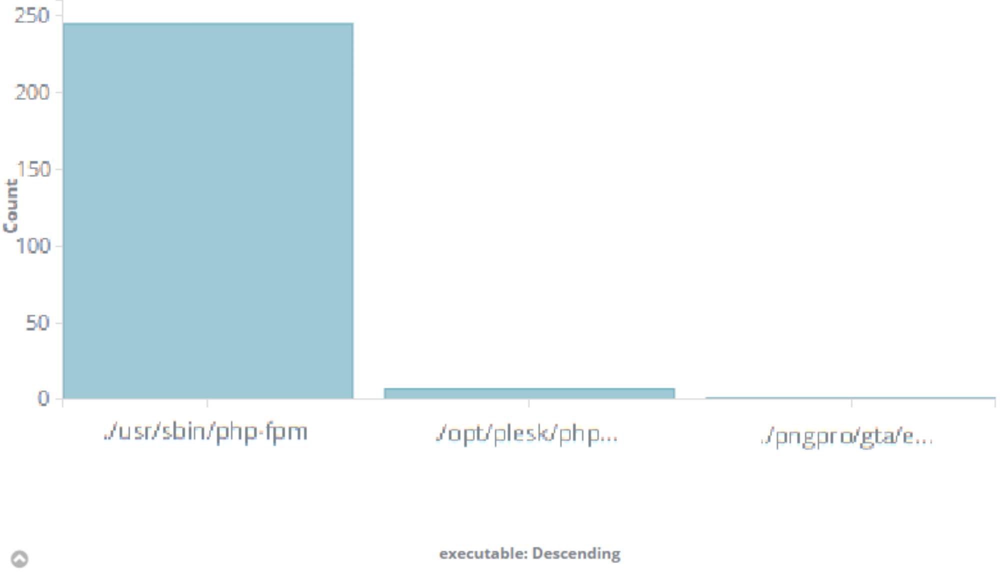


Figure 16: Example Visualization 2. Which application(s) caused the peak in crashes?

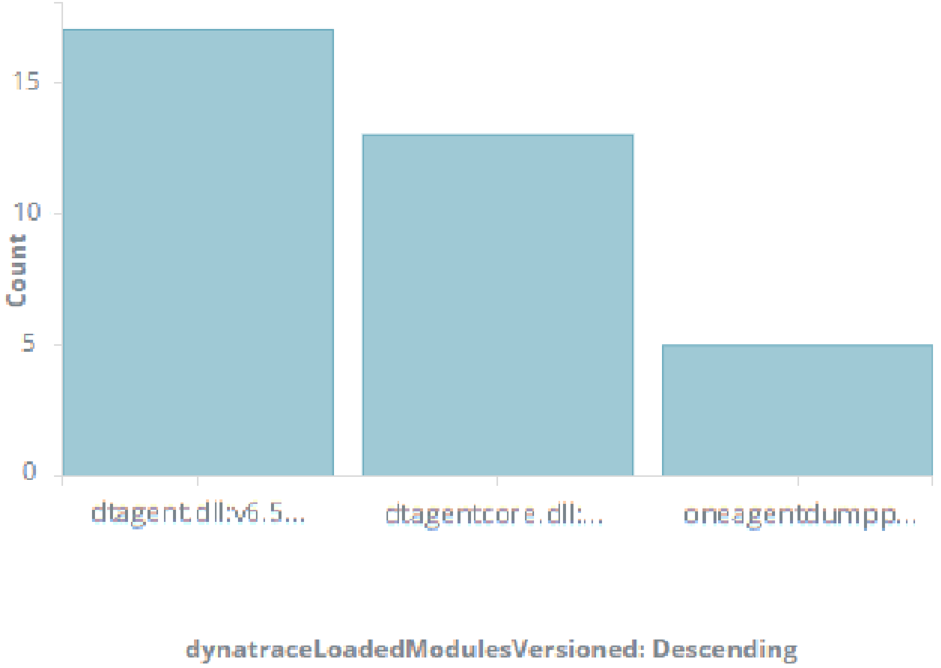


Figure 17: Example Visualization 3. Are there any shared libraries that tend to crash more often than others?

Similar to the answer provided for question 2, we can print the date histogram of the most frequently crashing modules. Setting the X-Axis to *Date Histogram* and adding a *Split Series* bucket with a *Terms* aggregation and the *dynatraceLoadedModulesVersioned* field selected, yields the crash frequency of modules over the according time frame. This chart can be particularly useful for checking if certain modules keep occurring over a long period of time or only occur in a short timeframe but with high frequency.

6.2.4 Limitations

As already indicated in the previous subsection, a critical problem is that the data only reflects crashing scenarios and neglects scenarios where the application was running successfully. This fact has to be kept in mind when checking visualizations that are based on crash dump data. A statement on the crash-likeliness of either executables or modules cannot be made with the data given.

A more subtle shortcoming is the missing timestamp of crashes. While there is a timestamp that refers to the time of analyzation, there is no way to know when the application actually crashed. This timestamp is not available in a core dump file. In the Dynatrace environment, this circumstance is usually not a concern because dumps are sent to the analysis service only with short delay. However, if the analysis service is not available for some time, core dump files must wait for analyzation until the analysis service becomes ready again. This may lead to incorrect peaks in the charts.

A third shortcoming are the differences between Windows crash dumps and Linux core dumps that may be problematic when creating visualizations covering both types of dumps. For example, Windows crash dumps do not store the executable field which may lead to undesired peaks in charts that show the number of crashes grouped by the executable. Also, some other fields have a slightly different meaning in Windows crash dumps, e. g., the *lastEventDescription* is a specific field in Windows crash dumps, while Linux core dumps merely store a textual description of the signal code.

7 Evaluation

This chapter provides a functional evaluation as well as a performance evaluation of the tool developed in this thesis. While the functional evaluation covers a feature comparison to a similar tool, the performance evaluation measures specific runtime metrics, such as memory utilization.

7.1 Functional Evaluation

Due to the lack of alternatives for automated core dump analysis, a direct feature comparison is not possible. The alternative to using SuperDump for core dump analysis is the manual analysis with a post-mortem debugger, such as GDB or DBX. GDB was chosen as the debugger of choice because GDB is the most common debugger for Linux based operating systems.

The feature comparison in Table 11 compares the features of SuperDump (SD) with GDB. Due to the sheer number of features of GDB, only the most relevant features for debugging Linux core dumps were taken into account. At this point, I want to emphasize that the objective of SuperDump is not to outperform GDB in any way. The main focus of SuperDump lies on usability.

7.2 Performance Evaluation

The system that is hosting SuperDump is a virtual 64-bit Windows Server 2016 running with 2 cores on 2 GHz each. The RAM limit of the VM is set to 8 GB. Besides SuperDump, there are no more processes running on that machine. The core dump analysis is executed in a Docker container.

The analyses that are presented in this section were executed each at a time, i. e., there were no interferences between analysis processes. There may still be interferences from other sources, such as network related problems, high load on the web server, or background services.

7.2.1 Analysis Time

The delay at which SuperDump can present results to users is critical as it affects its usability. If waiting for the results of a core dump takes too long, one of the major reasons to use SuperDump, namely its speedup for people who have to deal with core dumps, is erradicated. By automatically fetching core dumps from the issue tracking system, SuperDump can start analyzing much sooner than it could if each core dump

Feature	**GDB**	**SD**	**Notes**
Dump summary	✓	✓	
Process architecture	✓	✓	
User/Group ID	✓	✓	
Program entry point	✓	✓	
List of modules	✓	✓	
Mounting addresses of modules	✓	✓	
Version information for modules	✗	✓	SuperDump can extract version information from third party files
Automatic loading of debug symbols	✓	✓	SuperDump can also access remote repositories
Stacktraces of all threads	✓	✓	
Stack pointer	✓	✓	
Instruction pointer	✓	✓	
Return address	✓	✓	
Register contents	✓	✗	
Local variables	✓	✓	
Function arguments	✓	✓	
Global variables	✓	✗	
Source code view	✓	✗	SuperDump only provides a link to the repository
Highlight specific threads/frames	✗	✓	
List of auxiliary vector fields	✓	partial	SuperDump extracts only specific fields from the auxiliary vector

Table 11: Feature Comparison between GDB and SuperDump (SD)

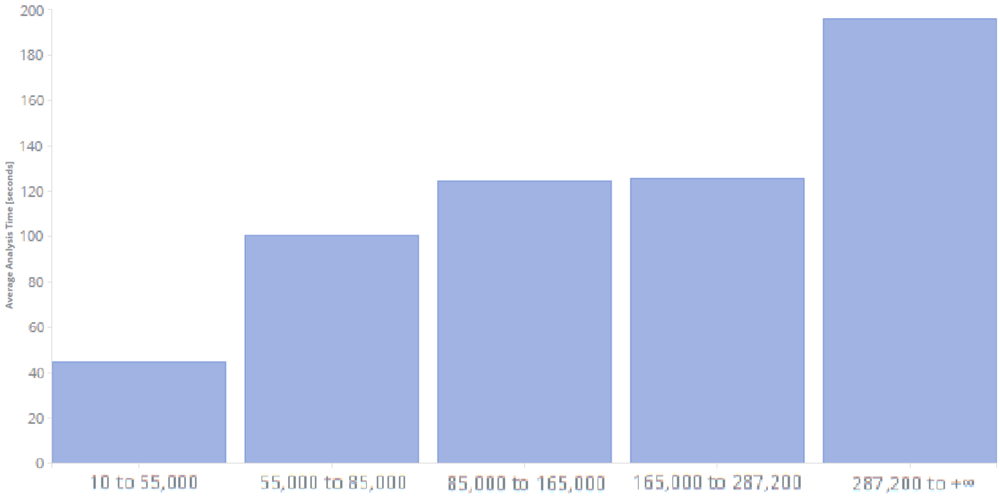


Figure 18: Average Analysis Time [seconds] grouped by dump size [kilobytes]

would have to be submitted manually. Therefore, the waiting time for users is highly reduced. Still, the analysis time must be kept low to reduce waiting times for users who want to view the results of recent core dumps. Furthermore, analysis times are crucial for manually submitted core dumps.

The analysis time, i. e., the time from starting the archive extraction until the final results are available, is stored for each core dump that was analyzed. Figure 18 shows the average analysis times for core dumps grouped by dump size in kilobytes. The analysis time is dependent on many factors. For example, caching the debug symbols speeds up analysis times for subsequent core dumps that share the same binaries. Another factor is dump size. Dumps that take more space on hard disk tend to require more analysis time. Therefore, a grouping by dump size was introduced in the analysis. The dump size buckets were chosen such that each group has approximately the same number of core dumps. The figure shows how the dump size affects the analysis time.

While the analysis times of the individual buckets range from 45 seconds up to 196 seconds, the total average analysis time is 123 seconds. For comparison, the average analysis time for Windows crash dumps is 108 seconds.

These average analysis times seem high considering the analysis steps described in Chapter 3. The main time consumer is the retrieval of stack information via GDB. In the Dynatrace environment, an average analysis time of about two minutes is satisfying. However, for applications that require faster analyses, there are two options to effectively decrease analysis times. Option 1 is to simply disable the stack information retrieval which might be a viable solution for certain applications. Option 2 requires implementing an algorithm to retrieve stack information and thereby circumventing the time consuming input/output operations of GDB. While disabling stack information retrieval decreases analysis times by approximately 80 percent, there is no estimate for the speedup provided by option 2.

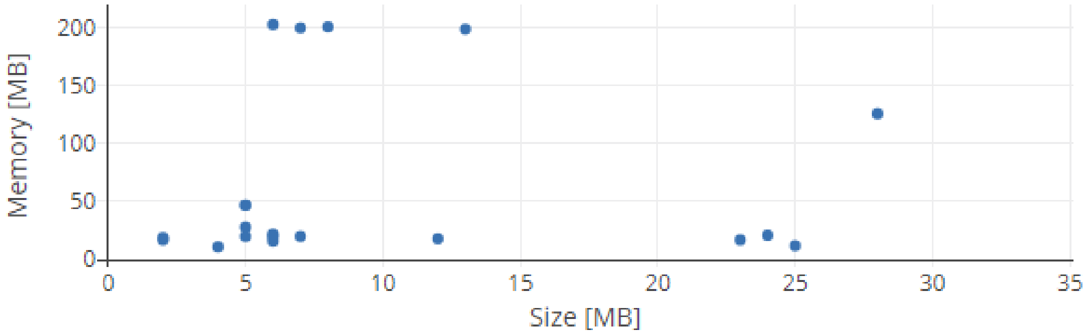


Figure 19: Memory usage in relation to core dump size

Dump Size	**# dumps**	**Min.**	**25th qtl**	**50th qtl**	**75th qtl**	**Max.**
up to 55 MB	20	11	17	20.5	126	203

Table 12: Memory usage statistics of 20 randomly selected core dumps

7.2.2 Memory Usage

Memory usage is an important factor when it comes to dimensioning the analysis system. If the memory setting does not suffice for the analysis, the system will begin swapping which lets analysis times soar.

Unfortunately, memory usage is not recorded during the automated analysis. In order to retrieve memory information, the analysis has to be started manually and memory consumption has to be recorded via an external tool, e. g., with the *docker stats* command. As core dumps are deleted after the analysis, the number of available core dumps which can be evaluated at this time is limited. In fact, all available core dumps have a size of 55 MB or less which makes a comparison between different core dump sizes impossible. Nevertheless, Figure 19 shows a scatter plot of the memory consumption in relation to the core dump size of 20 randomly selected core dumps from the live environment. The memory values were retrieved via sampling with the *docker stats* command which shows the consumed memory space of all running applications inside the container. The largest value from an analysis was used for the evaluation. Table 12 shows the according quantiles of memory consumption.

The scatter plot indicates that the core dump size does not have a major influence on memory usage. There are several small core dumps that require rather large amounts of memory, as well as there are large core dumps that suffice with little memory. Please note, that this evaluation is limited to core dumps with sizes of less than 30 MB.

8 Future Work

The core dump analysis tool implemented in this thesis provides useful information about core dumps to users but there is still room for improvement. Some aspects of potential improvements are discussed in following paragraphs.

Dynamic Tagging The tagging functionality described in Section 3.4 works nicely for static environments. The tags are statically defined in the source code. A better solution would be to allow users to define their own tags via the user interface. Furthermore, as the tagging only requires the finished result objects, the tags could be updated for core dumps that were already analyzed.

32-bit Dumps The problem of analyzing 32-bit dumps was discussed in Section 3.9. Upon the release of a 32-bit .NET core CLR for Linux, SuperDump would have to be updated to distinguish between 32 and 64-bit core dumps.

Analysis Report Summary on JIRA Tickets Currently, users of the issue tracking system JIRA need to follow the analysis link posted by SuperDump in each ticket with a core dump. If SuperDump could post a summary of the results on the ticket via a comment, users could immediately get an overview of the core dump contents. This summary may contain fields like the crashing process, call arguments, and the signal code.

Optimize Docker Images The docker images that were generated in the scope of this thesis are not well optimized. Using more compact base images, e. g., by reducing the number of explicit *RUN* commands could reduce the size of the images. The problem of large images is not only the disk space consumption but also the increased deployment times when the images are created.

Duplication Detection The live environment that is running for six months has shown that many core dumps are very similar and differ only in minor details. When viewing the results of a core dump, users might want to know about similar crashes. For example, a crash that occurred many times in the past is more critical than a crash that occurred just once. The difficult part of this improvement is to invent an algorithm that accurately detects duplicated core dumps.

Automatic Peak Detection in Charts The charts presented in Section 6.2 are well suited for detecting anomalies in crash occurrences. Currently, this anomaly detection has to be done manually. An automated approach would be more practical. For example, the tool could notify a designated SuperDump user if a specific application crashed more than ten times in a single week.

9 Conclusion

The Linux core dump analysis tool developed in this thesis is a powerful tool for developers who get confronted with core dumps from time to time. Although there a several similarities to conventional post mortem debugging techniques, SuperDump is aimed at a different audience. Whereas conventional debuggers excel in providing all kinds of data stored in a core dump, SuperDump focuses only on the most critical data and presents it in a user friendly way. The feature comparison with GDB lists several types of information that cannot be retrieved with SuperDump but with GDB. Instead, SuperDump comes with a much more user friendly interface. For example, as opposed to GDB, SuperDump is able to highlight specific threads which improves the navigation between threads. Furthermore, SuperDump is fully automated and can be integrated into the development life cycle. With the interactive mode, users can start a live GDB debugging session within their web browser allowing for a more sophisticated core dump analysis.

The most crucial shortcoming of SuperDump is the missing support for 32-bit core dumps. A .NET core framework for 32-bit systems is currently being worked on which would resolve this issue. Another disadvantage is the rather sophisticated input format of core dumps. The core dump file has to be packed into an archive along with all shared libraries. The files must comply to the exact same structure as on the system that the process was running on. Creating such an archive manually requires a considerate amount of work especially if many shared libraries were involved.

The collected data provides useful statistics for anomaly detection for Linux core dumps as well as for Windows crash dumps. Although there are a few restrictions, the generated data gives developers the opportunity to check the dump history for trends or peaks.

Finally, the evaluation has shown that SuperDump suffices with a low amount of memory and is able to process the majority of core dumps in less than three minutes. While three minutes may seem long for an automated analysis, it is more than enough for the Dynatrace environment. More time critical environments could disable the stack information retrieval resulting in a speedup of approximately 80 percent.

SuperDump has proved successful to Dynatrace developers. With the integration into the development life cycle, application crashes are analyzed automatically as opposed to the manual approach of using command-line debuggers. In most cases, the information displayed in a crash report is sufficient to reveal the root cause of the crash which removes the burden of using in-depth debugging tools completely. The remaining crashes that are too complex to be automatically analyzed, can be covered with the interactive mode, i. e., a live debugging session within the web browser. Therefore, developers can access the full power of in-depth debugging tools without having to bother about any setup.

Further, the statistical evaluation gives insights into the history of crash dumps. Crash dump anomalies that would be hardly noticeable without tool support can be detected immediately from the Kibana dashboard which can be tailored to the exact needs of the user. The detection of such anomalies leads to faster treatment of potential bugs and therefore improves software quality.

List of Figures

List of Tables

Listings

Bibliography

[1] Gnu Debugger (GDB). https://www.gnu.org/software/gdb/. [Online; accessed 09-August-2017].

[2] IBM. dbx Debugger. https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.2.0/com.ibm.zos.v2r2.bpxa500/bpxa50021.htm. [Online; accessed 09-August-2017].

[3] Oracle. Modular Debugger. https://docs.oracle.com/cd/E19455-01/806-5194/6je7ktfkb/index.html. [Online; accessed 09-August-2017].

[4] LLVM Developer Group. The LLDB Debugger. http://lldb.llvm.org. [Online; accessed 09-August-2017].

[5] Arnaud Desitter. Comparison between Sun dbx and gdb. http://www.fortran-2000.com/ArnaudRecipes/CompGdbDbx.html. [Online; accessed 09-August-2017].

[6] Jim Mauro Richard McDougall. The modular debugger. In *Solaris Internals, Second Edition*, pages 123–218, July 2006.

[7] Dynatrace Austria GmbH. SuperDump Github. https://github.com/Dynatrace/superdump. [Online; accessed 09-August-2017].

[8] UNIX System Laboratories (USL). ELF Specification. http://www.skyfree.org/linux/references/ELF_Format.pdf. [Online; accessed 09-August-2017].

[9] SCO ELF Header specification. http://www.sco.com/developers/gabi/latest/ch4.eheader.html. [Online; accessed 09-August-2017].

[10] The Open Group. The Open Group Base Specifications Issue 7. http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/signal.h.html. [Online; accessed 09-August-2017].

[11] Docker Inc. Docker. https://www.docker.com. [Online; accessed 09-August-2017].

[12] Trisha McCanna. Considerations for running Docker for Windows Server 2016 with Hyper-V VMS. https://blog.docker.com/2016/10/considerations-running-docker-windows-server-2016-hyper-v-vms. [Online; accessed 09-August-2017].

[13] Michael Friis. Docker on Windows Server. `https://blog.docker.com/2016/09/build-your-first-docker-windows-server-container`. [Online; accessed 09-August-2017].

[14] Algore. Running Linux containers on Windows Server 2016. `https://forums.docker.com/t/linux-container-in-w2k16/26321/2`. [Online; accessed 09-August-2017].

[15] Docker Inc. Docker Compose Documentation. `https://docs.docker.com/compose`. [Online; accessed 09-August-2017].

[16] Elasticsearch. Elasticsearch. `https://www.elastic.co/de/products/elasticsearch`. [Online; accessed 03-January-2018].

[17] David Mosberger. Libunwind. `http://www.nongnu.org/libunwind`. [Online; accessed 09-August-2017].

[18] Arun Sharma. Libunwind 1.1 release changelog. `http://savannah.nongnu.org/forum/forum.php?forum_id=7392`. [Online; accessed 09-August-2017].

[19] getauxval manual. `http://man7.org/linux/man-pages/man3/getauxval.3.html`. [Online; accessed 09-August-2017].

[20] Denys Vlasenko. NT_FILE extension to Linux kernel. `https://patchwork.kernel.org/patch/1481921/`. [Online; accessed 09-August-2017].

[21] addr2line manual. `https://linux.die.net/man/1/addr2line`. [Online; accessed 09-August-2017].

[22] Iwasaki Yudai. GoTTY GitHub. `https://github.com/yudai/gotty`. [Online; accessed 09-August-2017].

[23] Docker Inc. Docker Hub. `https://hub.docker.com`. [Online; accessed 09-August-2017].

[24] Elasticsearch. Kibana. `https://www.elastic.co/de/products/kibana`. [Online; accessed 03-January-2018].

[25] Grafana Labs. Grafana. `https://grafana.com`. [Online; accessed 03-January-2018].

[26] Asaf Yigal. Comparison between Kibana and Grafana. `https://logz.io/blog/grafana-vs-kibana`. [Online; accessed 03-January-2018].

[27] The Apache Software Foundation. Apache Lucene Query Syntax. `https://lucene.apache.org/core/2_9_4/queryparsersyntax.html`. [Online; accessed 03-January-2018].

Dominik Steinbinder

Resume

Personal Profile

Date of Birth	**22nd August, 1992**.
Nationality	**Austria**.
Address	**Michael-Hainisch-Straße 6 / 2** **4040 Linz**.

Work Experience

since 10/2014	**Software Developer**, *Dynatrace Austria GmbH*, Linz.
07/2014–09/2014	**Internship as Software Developer**, *Dynatrace Austria GmbH*, Linz.

Education

Academic

since 09/2016	**Master Studies in Computer Science**, *Johannes Kepler University Linz.* Major Subject: Software Engineering
10/2012–08/2016	**Bachelor Studies in Computer Science**, *Johannes Kepler University Linz.*

School

09/2006–07/2011	**Higher Technical Education Institute**, *Braunau am Inn.*

Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references. This printed thesis is identical with the electronic version submitted.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Ort, Datum	Dominik Steinbinder