Automating Linux Post-Mortem Debugging
Abstract

Post-mortem debugging is often the last bastion of debugging solutions. It involves analyzing a raw memory dump of either a portion of memory or the whole memory of the system at an instance of time, much like a photograph. In order to able to read such a memory dump, you have to have knowledge about how the system was built and implemented. Some systems, especially embedded ones, are the ones that can benefit the most from it, due to other solutions being impractical or simply unavailable. The process of post-mortem debugging is cumbersome, as it requires a lot of file operations in order to be executed. The automation of post-mortem debugging can potentially increase its effectiveness greatly.

This research was conducted at a large-scale Finnish telecommunications company, which uses embedded Linux devices in its products. The objective of this research was to provide a solution for automating Linux post-mortem debugging, which is better than the current solutions and workings available in the case company. The research method used is the acclaimed design-science research.

The research justifies its approach by looking at related work exploring the benefits of automated debugging in general. It also looks at common trends and pitfalls when dealing with highly complex specialized embedded processing environments. The design science artifact, created by this research, was made by combining the justification from the related work, combined with the requirements and processes of the case company environment. The evaluation of the research, both quantitative and qualitative, explores its benefits and drawbacks from multiple points of view.

Overall, the implementation of the design science artifact, created by this research, was a success. Its time-saving capabilities were clearly demonstrated, as well as its potential use as a learning tool for new hires and non-technical staff. The main drawback identified was the need of constant maintenance, which is common for applications with great variability in their environment.

Keywords
automation, embedded, debugging, corefile, telecoms, Linux, Python

Supervisors
University of Oulu - Ph.D., University Lecturer Antti Siirtola
Free University of Bolzano - University Lecturer Andrea Alexander Janes
I would like to thank all of my friends and family without whom I would have finished this project much sooner.
Abbreviations

BSD = Berkeley Software Distribution
CDT = C/C++ Development Tooling
ELF = Executable and Linkable Format
ext4 = Fourth Extended File System
GCC = GNU Compiler Collection
GDB = The GNU Project Debugger
GUI = Graphical User Interface
JFS = Journaled File System
OS = Operating System
XFS = Extended File System
Contents

Abstract .......................................................................................................................... 2
Foreword ......................................................................................................................... 3
Abbreviations .................................................................................................................. 4
Contents .......................................................................................................................... 5
1. Introduction ................................................................................................................ 7
  1.1 Linux core file ......................................................................................................... 7
  1.2 Linux journal file .................................................................................................... 8
  1.3 Research environment ............................................................................................ 8
2. Research problem and method .................................................................................. 9
  2.1 Research questions .................................................................................................. 9
  2.2 Design Science Research ....................................................................................... 9
    2.2.1 Design Science Guidelines ............................................................................. 9
    2.2.2 Design Science Research Framework ............................................................ 11
3. Related Work .............................................................................................................. 13
  3.1.1 Benefits of automated debugging ..................................................................... 13
  3.1.2 Benefits of automated debugging on embedded systems ............................... 13
  3.1.3 Specialized processing environment challenges .............................................. 14
4. Implementation .......................................................................................................... 15
  4.1 Input files .............................................................................................................. 15
  4.2 Case company environment .................................................................................... 16
  4.3 Iterative design ...................................................................................................... 16
  4.4 Main modes of operation ...................................................................................... 17
    4.4.1 Setup/Jenkins mode ........................................................................................ 17
    4.4.2 GDB mode ....................................................................................................... 20
    4.4.3 Eclipse mode ..................................................................................................... 20
  4.5 Environment Control and Environment Parameters ............................................. 21
  4.6 Folder structure ..................................................................................................... 22
  4.7 Input parameters .................................................................................................... 23
    4.7.1 Input files ........................................................................................................ 23
    4.7.2 Mode of operation ........................................................................................... 23
    4.7.3 Type of product family .................................................................................... 24
    4.7.4 Version overrides ............................................................................................ 24
    4.7.5 Custom shared libraries override .................................................................. 24
    4.7.6 Extra GDB commands for analysis ............................................................... 24
  4.8 Input format classifier ............................................................................................ 25
  4.9 Debug libraries module .......................................................................................... 25
  4.10 Custom libraries module ....................................................................................... 26
  4.11 Shadow system root module ............................................................................... 26
  4.12 Context file .......................................................................................................... 27
  4.13 Analysis modules .................................................................................................. 27
    4.13.1 Journal Analyzer ............................................................................................ 27
    4.13.2 Memory logs Analyzer .................................................................................. 28
    4.13.3 Corefile Analyzer ......................................................................................... 28
  4.14 User Stories .......................................................................................................... 29
5. Evaluation .................................................................................................................. 32
  5.1 Time metrics .......................................................................................................... 32
    5.1.1 Summary of time metrics .............................................................................. 36
  5.2 User stories ............................................................................................................ 37
  5.3 Semi-structured interview comments ..................................................................... 37
5.3.1 Interview environment and interviewee information .......................... 37
5.3.2 Limitations of the interviews...................................................... 38
5.3.3 Interview results ......................................................................... 39
5.3.4 Summary of interview results...................................................... 43
6. Discussion and design science guidelines evaluation ................................ 44
   6.1 Evaluation of design science research guidelines.......................... 46
7. Conclusion.......................................................................................... 48
   7.1 Limitations.................................................................................. 48
   7.2 Future work.................................................................................. 48
References............................................................................................. 50
Appendix A. Semi-structured interview form........................................... 53
1. Introduction

Post-mortem debugging involves analyzing a memory dump of a portion of the memory of the system at an instance of time. Usually, this memory image is taken when the system has crashed or has terminated abnormally. The memory dumps can be full or incomplete and can hold different memory sizes. These memory dumps are basically a snapshot of the memory of a system, akin to taking a photograph at a given moment, indicating the state of the system. Almost all types of operating systems provide a way to obtain such a memory snapshot (Glerum et al., 2009 October).

All of these snapshots will be different from each other, due to the way that the operating system manages its memory. This thesis will be focusing on Linux (The Linux Foundation, 2018) post-mortem debugging specifically, as this is the operating system used in the case company the research is conducted at. Linux memory dumps are usually referred to as core dumps or core files.

The objective of this master’s thesis is to look at the overall effects of implementing an automated post-mortem debugging solution. This type of debugging is usually the first line of defense for many industries that have embedded hardware deployed worldwide, which has no easy debugging access. Currently, post-mortem debugging takes a fair share of time to do, as there a lot of variables that need to be taken in account to successfully interpret the data generated during a system crash.

In the following sections of the introduction, the research will discuss what kind of data is used, in what environments it is used (case company), and why is it an extremely useful way to debug certain systems.

1.1 Linux core file

Linux core files or core dumps are files containing the memory space of when a process terminates unexpectedly. Core file generation can be triggered manually or automatically. Core dumps are generated by the Linux kernel automatically when a program crashes (Kerrisk, 2018). This is functionality common to all of the UNIX-like operating systems. By default, the core files are in a standard executable image format for Linux – for older formats a.out and for newer versions in an ELF (Executable and Linkable Format) (Drepper, 2006), which is common for all modern Linux, Solaris and BSD (Berkeley Software Distribution) systems.

In order for the core files to be readable, one needs to know how to read them, as the file contents are in a hexadecimal format and the memory locations are not static (Bligh, Desnoyers & Schultz, 2007 June). Core files are generally interpreted with the use of GDB (The GNU Project Debugger) (Free Software Foundation, Inc., 2018b), which in turn uses symbol tables to interpret the raw memory dump. The symbol tables or files are created during the compilation of a program. They represent the data structure used by a software language compiler or interpreter (for non-compiled languages). In this case, the Linux kernel is written in C and assembly and is compiled by the GCC (GNU Compiler Collection) compiler (Griffith, 2002). During the compilation stage, GCC creates the executable files and the symbol files needed to interpret the memory.

When compiling the software, GCC can be set to produce additional debug symbols, which can be used to decode more information from the executable binary and memory
dump. Such extra information can be in the form of source code lines and file names of
the associated memory, local variable information of the snapshot during run-time and
names of various identifiers. These extra debug symbols can be coded with the executable
binary, but are often discarded altogether or stored somewhere else separately. These
debug symbols can come in the form of a shared library or an executable. The executables
get executed on the system as the name suggests and the shared libraries represent code
that can be used and reused by multiple processes on the system. Most of the Linux
software development is done using shared libraries and executables (Jones, 2018a).

Briefly, in order to be able to read and interpret a core file, you need to have the
executable, the shared libraries used in it and the debug symbols generated during the
compilation stage associated with the processes running in the core file (Zhang & Wang,
2003).

1.2 Linux journal file

Another useful source of data when it comes to post-mortem debugging is the Linux
journal file. Typically, Linux is used with a journaling file system - ext4 (Fourth Extended
File System), JFS (Journabled File System), XFS (Extended Files System) (Jones, 2018b).
A journaling file system keeps track of all committed and uncommitted of the changes
happening to the system. The data structure that holds this information is usually called
the “journal”, hence the Linux journal file name. The Linux journal is a circular log,
which holds the current state of the system and the recent changes (Zhang & Wang, 2003).

In the case of this paper, the journal file is obtained at the same time as the core files,
therefore the journal file can shed more light about the state of the system during the crash
(when the core file gets generated) (Galli, 2001).

The most useful information for post-mortem debugging located in the journal is the stack
trace of each processor that was running on the system during the crash (Bligh, Desnoyers
& Schultz, 2007 June). When the system starts the emergency shutdown during a crash,
each processor’s stack is saved to the journal file and can be inspected later. The crash
stack holds information for all of the processes running on a certain processor and their
memory information (Zhang & Wang, 2003). That raw memory information can be
decoded in a similar way to the core files explained in Section 1.1.

1.3 Research environment

This research is conducted at a large-scale telecommunications company and will be
tailored to the company’s working environment and requirements. The requirements are
sufficiently broad that multiple ways of achieving the desired end result can be explored,
as the company’s limitations do not have catastrophic effects or major influence on design
choices to be made. The main purpose of the tool is to simplify and speed up the process
of post-mortem debugging by the software developers (technical role) and by the testing
staff (semi-technical role). The main product of the company is a line of real-time high-
processing embedded systems used globally.

The current solution in place is not mature at all and it takes moderate knowledge of the
system in the case company’s environment and knowledge of the tools used in the
solution.
2. Research problem and method

This chapter will talk about the research problem and method, by defining research questions. The research method defined will be explained and its of its guidelines and their applications will be further described.

2.1 Research questions

The objective of this is research is to provide a solution for automating Linux post-mortem debugging, which is better than the current solutions and workings available. In order to be able to judge the effectiveness of the new solution, it has to be compared to what was set in place before it in the environment under research. The implementation of this tool is done at a large-scale telecommunications company, which has no automated post-mortem debugging tools. The strength of the impact and types of workers affected will be judged against the previous solution, which was manually doing the post-mortem debugging.

Main Research Question: How can an automated solution for Linux post-mortem debugging be implemented?

Research Question #1: How much time does an automated solution save, as opposed to doing the process manually?

Research Question #2: How does the amount of pre-knowledge required differ from the automated solution, as opposed to the manual one?

By answering the 1st and 2nd research questions, we can gain more insight on the main research question. By looking at the time required to start debugging and the technical capability necessary for being able to do the debugging the main research question can be answered adequately.

2.2 Design Science Research

The research method used in this paper is the design-science research. It focuses on the development and performance of designed artifacts, by trying to improve the artifact’s performance (Hevner, March, Park & Ram, 2004). The research method fits this research topic well, because there is a specific problem set in a case company environment. The new solution (artifact) will be compared versus the old solution used in the case company, which will serve as a good source of metrics and information for the evaluation. The guidelines and framework of this type of research will be described in the sub-sections below.

2.2.1 Design Science Guidelines

The fundamental principle of design-science research is that knowledge regarding a certain problem or design is gained by building and applying the artifact developed (Hevner, et al. 2004.) The definition of guidelines that need to be followed to complete this type of design-science research are shown in Table 1. Each of the guidelines and its relevant counterpart information will be described below.
Table 1. Design-science research guidelines that are followed in this study adapted from Hevner, et al., 2004

<table>
<thead>
<tr>
<th>Guideline</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guideline 1: Design as an Artifact</td>
<td>The design-science research must produce an artifact or a solution to the relevant problem.</td>
</tr>
<tr>
<td>Guideline 2: Problem Relevance</td>
<td>The objective of the design-science research is to develop a technology-based solution to problems relevant in the industry.</td>
</tr>
<tr>
<td>Guideline 3: Design Evaluation</td>
<td>The developed design artifact must be scrutinously evaluated and its purpose must be clearly demonstrated with rigorous evaluation methods.</td>
</tr>
<tr>
<td>Guideline 4: Research Contributions</td>
<td>The design-science research must rely on and provide renown contributions in the areas relevant to the design artifact, design methodologies and foundations.</td>
</tr>
<tr>
<td>Guideline 5: Research Rigor</td>
<td>Design-science research requires the application of strictly adhered to processes in the implementation and the evaluation of the design artifact.</td>
</tr>
<tr>
<td>Guideline 6: Design as a Search Process</td>
<td>The identification of an effective artifact solution requires any the use of available methods to reach the desired result, while adhering to the local environment requirements.</td>
</tr>
<tr>
<td>Guideline 7: Communication of Research</td>
<td>The design-science research must be presented in such a way that satisfies both technologically and management-oriented audiences.</td>
</tr>
</tbody>
</table>

**Guideline 1: Design as an artifact**

The artifact in question will be the final automated post-mortem debugging tool. There were no official requirements set during the initial steps of development, except that the artifact needs to be able to provide useful extracted debugging information.

**Guideline 2: Problem Relevance**

The problem solved in this scope is the automatic extraction of beneficial information from Linux core memory dumps. The business value of this is the increased efficiency when debugging memory dumps and access to debugging techniques for non-technical people.
Guideline 3: Design Evaluation

The business environment, in this case, the case company establishes the requirements upon which the artifact will be evaluated against. There is no clear universally defined metric, which the tool can be tested against. The utility and efficacy of the artifact will be demonstrated by comparing the time taken to run the same type of debug process when doing it manually and when using the tool. Feedback and comments, in the form of a semi-structured interview, will also be included. The evaluation will also include comments and feedback from involved users in the form of a semi-structured interview. This method can be also called observational. The observational method deals with the study of the artifact in the business environment and its stakeholders.

Guideline 4: Research Contributions

The most important contribution of this research will be the artifact itself, as the knowledge required in the design process is widely known and accepted. By solving the problem at hand, there will be a clearly presented solution applying the existing scientific knowledge in a business environment.

Guideline 5: Research Rigor

In this case, research rigor refers to the way in which the research is conducted. As the environment in which this research is conducted cannot be very formalized, the rigor will be assessed with respect to its applicability and generalizability.

Guideline 6: Design as a search process

The development of this artifact highlights design as a search process very well, due to the minimal requirements set at the initial steps of development. The ultimate goal is to create an automated debugging system, therefore there is a lot of freedom to explore the best-fit solution. The search process is supplemented by applying knowledge from existing scientific literature. As the artifact is not proposing a mathematically standard technique, it is unfeasible to go through every possible infrastructure iteration, therefore there cannot be an optimal solution found. Instead, the solution will be tailored to the environment in which it works.

Guideline 7: Communication of Research

This research is mainly aimed at a technology-oriented audience, by describing the artifact creation in detail. Nonetheless, it can also be used for a management-oriented audience as the organizational resources and processes required are described as well.

2.2.2 Design Science Research Framework

The design science framework follows the conceptual recommendations set by Hevner et al. (2004). The use of such a framework allows for easier understanding, comparison and evaluation of the research. Figure 1, illustrated below, shows the framework visually.
Knowledge Base: The closest we can get to a real knowledge base is familiarity with tools most adequate to this job and the limitations present in the case company environment. The most important tool in use is GDB and its capability of interpreting core memory dumps. With GDB handling the central part of the debugging processing, there needs to be a system built around supplying and returning information to/from the artifact in development. In this case, it is extremely useful to follow general automated debugging methodologies and intra-company communication and collaboration tools.

IS Research: The ultimate purpose of this research is to develop and implement a working variant of a system, which can automatically extract and perform debugging analysis on key files present in the supplied information. The focus of this tool is on the post-mortem debugging aspect, but the functionality can be greatly added upon if necessary, as the software infrastructure and case company environment connections will be already present. Evaluating the artifact will be achieved by conducting semi-structured interviews with stakeholders from within the case-company, as there are no formal requirements set to this project. The qualities most focused on in the evaluation will be the artifact’s benefits in regards to how it improves the overall process of conducting post-mortem debugging.

Environment: This research is conducted at a large-scale telecommunications company; whose products operate on a global scale successfully. The information processed by the artifact is collected from the products operating either in a test environment or in a real world setting. The products’ architecture can be described as a high-processing real-time embedded system with high software and hardware complexity. The information from the products (to be analysed by the artifact) is collected in the form of snapshot logs, which contain information about the products’ state at the time of the data collection. The logs are significant in size and only a small part of the information contained in them is actually analysed by the artifact developed in this research. The products are mainly developed in C/C++ and the testing environment is mostly driven by Python (Python Software Foundation, 2018d). As the main focus on this research is not on the case company product development, but on debugging post-mortem data, only technologies relevant to this scope will be taken account of.
3. Related Work

This chapter considers existing knowledge and identified problems about the foundations and methodologies identified in Section 2.2.2, and information regarding the general software and hardware ecosystem related to the case company this research is conducted in.

3.1.1 Benefits of automated debugging

Debugging is necessary when a problem has occurred and its origin is still unknown. This means that the problem was not identified during testing or any other quality check before deploying the software. By estimations from globally collected data, almost half of the resources allocated to any software project are used for software testing (Myers, Sandler & Badgett, 2011). Given this impressive resource allocation for testing, it is sensible to increase the focus on other forms of error-finding practices, such as post-mortem debugging. As the testing process can never be perfect and there will always be unforeseen faults (Weinberg, 2008), even slightly investing in post-testing analysis or debugging can be extremely cost-effective based on the volume of already allocated resources for testing.

Debugging tools work best when they integrate the different activities in debugging and provide a complete ecosystem for debugging. It also paves the way for better usage of “richer” information (test cases, values) to make the debugging aid more usable in the future (Parnin & Orso, 2001 July). As the environment, this research is conducted in, is a global mature company with established processes, this is highly relevant to the problem scope (Coplien, Hoffman & Weiss, 1998).

Debugging also takes significant human resources to achieve (Lwakatare et al., 2016) and any action that decreases the need for human involvement can greatly improve the workload of the software project.

3.1.2 Benefits of automated debugging on embedded systems

Embedded systems typically require specialized knowledge, run on specialized hardware and have unique constraints, which decreases the availability of commonly available and acclaimed debugging tools (Lwakatare et al., 2016). This fact makes the use of already available debug information more attractive and lowers its associated costs compared to developing a high-level case-by-case custom tailored debugging solution.

Software cycles for embedded systems usually involve extensive code changes over fewer releases. This kind of release format makes it more difficult to pinpoint failures to certain code changes, which diminishes the effectiveness of conventional debugging methods (Kopetz, 2011).

Pursuing this approach also opens possibilities for future enhancement on debugging of core files and raw memory dumps. If there is an already set-up debugging ecosystem, there are many paths to take in order to include extra debug information for little-to-no processing and development cost, while drastically increasing the readability of the debug extracts (Kuzara, Blasciak & Parets, 1995).
With the required high degree of specialization, there are people personally responsible for certain modules or part of modules and those developers often come from different backgrounds (Kopetz, 2011). In turn, this creates a more complex development, distribution and deployment environment. Using resources already present and extractable is not greatly affected by the rising complexity of the software ecosystem.

There is an identified trend that development for embedded systems and specialized processing setups rely more and more on programming models and development frameworks. These kinds of environments follow a similar algorithmic structure and programming practices, which allows for greater reusability of tools created for that specific environment (Myers et al., 2011). If the reusability is higher, the cost of development needed to reach the whole organization is less.

Embedded systems have long up-times and the size of information required for typical debugging techniques and can be huge (Huselius, Sundmark & Thane, July 2003). Collection of such data usually has an impact of the system’s processing power as well. This makes it impractical or even impossible to store all the system’s history and this is where post-mortem debugging comes in. By limiting the stored or extracted information to a snapshot or a memory dump of the system in its undesired state, there is no need for large storage or the diminishing of performance due to data collection and storage.

3.1.3 Specialized processing environment challenges

In 2004, the single-processor performance increase started to slow down considerably. Consequently, every high-performance processor family in the embedded world and outside is moving to multiprocessor designs and multiple physical processors (Engblom, 2007 April). With this kind of a specialized environment, a lot of the usual debugging techniques and present tools fall short, creating the need of making a custom debug setup in-house (Hoyecki, 2009).

In order to specifically analyze each processor of a specialized multi-processing environment it is required for each instance of a debugger to have its own connection (Hoyecki, 2009). There are available tools and techniques, but post-mortem debugging provides good, quick and accurate information of the base system and its running processes (Huselius et al., July 2003). It is a good point to focus on as it maximizes utilization of already available information, instead of trying to create separate debugging connections or frameworks in another medium.

There is an identified need of good interoperability between general purpose processors and specialized processors (Sidler & Eguro, 2016 December), as most of the faults often occur in the points where the processors communicate between themselves. Using post-mortem debugging and raw memory maps can help create a very useful high-level overview of system processing flow and can help identify fault trends or weak links in the software.
4. Implementation

This section explains the environment and justification for the choice of technology used, the main use cases of the artifact and relatively high-level explanations using block diagrams. Finally, gathered (from the case company) user stories are presented in order to clearly illustrate the desired functionality and the evaluation requirements.

4.1 Input files

The artifact’s main purpose is to automatically analyze and obtain information from pre-supplied input files. These input files vary in size and hold massive amounts of information regarding the system to be debugged. Only a small portion of each input file is directly relevant to the desired debugging process.

There are multiple formats of each input file, depending on the processor used in the product or the product itself. For the purpose of this paper only the most common formats will be discussed.

A typical input file or commonly called a ‘snapshot’, consists of hundreds of nested .zip archive files. Each of those nested .zip files contains certain information about the product. Only a specific nested .zip file and certain .xml files are useful for the debugging process and need to be extracted. The format is semi-dynamic and there is not a known pre-determined location, which can be relied upon. Therefore, the artifact needs to be able to intelligently read the .zip input file and determine the location of the required files to be used for the debugging process.

The corefiles, Linux journals, the in-house software build version and other supporting debugging files are then extracted from one or more of the nested .zip files located in the input file itself.

Briefly, there can be multiple formats of the input ‘snapshot’ for different product families in the case company. There can also be multiple formats for each product family itself, which leads to several possible combinations of the ‘snapshot’ information folder structure. The figure below illustrates these possible combinations.

![Input or 'snapshot' possible formats](image)

Figure 2. Possible input formats that the artifact can receive

As seen in Figure 2, Formats 1.1 and 1.2 etc. are named similarly due to their shared commonality, although they are not exactly identical and need to be treated differently in the artifact.
4.2 Case company environment

The environment this research is conducted is an already mature environment with established tools, software technologies, and processes. This has influenced the choice of technology in constructing the artifact greatly. This sub-section will describe the tools and processes used in the environment and will try to justify the choices made.

The artifact itself is made entirely in native Python 2.7.3+. Native meaning no uncommon modules obtained from outside of the official Python package provider and 2.7.3+ covers all of the version of Python after the Python 2 and 3 fork (Wichmann, 2017). The choice of Python fits the purpose of this artifact very well, due to possibility of its use under different operating systems and systems with greatly varying specifications. Another point for Python is that it does not require compilation, which simplifies the storage of the code of the artifact itself and its deployment. The omission of a compilation stage removes the requirement of storing binaries, eliminates the major differences between different system environments, and facilitates the maintenance of the code.

The artifact is also used within a web service available to all of the engineering staff of the case company. There is an already established use of Jenkins (Jenkins Project, 2018) for a multitude of tests and other in-house supporting services. Integration of the Python artifact under Jenkins is straightforward and does not require any additional modification for that specific facet. The use of the artifact within a Jenkins environment is further described in Section 4.3.1.

The artifact can be used on Windows and Linux operating systems, and Python is already present and tested on all such systems used in the case company. The choice of using only ‘basic’ Python modules is due to the possible bloating that could occur during the future maintenance of the artifact. Using only native Python module assures that even systems that have not been previously set-up to for use with the artifact can run it without any expert modifications or any unforeseen software compatibility issues that may arise.

4.3 Iterative design

The artifact was designed using an iteration process, which culminated with the implementation described in this whole section. The beginning of the development was trying to mimic the previously existing process completely and served as an initial base of discussion between the stakeholders and the developer of the artifact.

The discussions about the iterations of the artifact took place in the case company environment and could be described as technical meetings. This type of meeting is fairly common in the software engineering world (Herbsleb, 2007 May). The sole topic of the meeting was the design artifact and any technical details concerning its integration with the environment of the case company. The meetings took place every 2-3 weeks and took roughly 1 hour each with varying numbers of interested stakeholders. The prototype of the artifact was demonstrated at each of the meetings and its results, and architecture were discussed. The stakeholders involved in these meetings were from different backgrounds – from team managers to software test engineers. The comments from the stakeholders, obtained from these meetings, were taken into account when developing the artifact and served as a way of aligning the supported functionality with the desired end result.
The purpose of these meetings was, ultimately, the continuous involvement of stakeholders with the continuous development of the artifact. The meetings were used as a way to collect user stories (See Section 4.14 User Stories), with which the desired functionality of the artifact was described.

4.4 Main modes of operation

There are three main views or modes of operation of the artifact

1. Setup/Jenkins mode – analyzing and extracting the ‘snapshot’
2. GDB mode – analyzing a specific corefile after setup mode has been run once
3. Eclipse mode – similar to GDB mode, but using Eclipse CDT (C/C++ Development Tooling) to analyze the corefile (Eclipse Foundation Inc., 2018)

These modes and the external inputs they take are controlled by passing command line arguments when launching the artifact. The specifics of the command line arguments are explained in more detail in Section 4.5. The purpose of this section is to give a high level explanation and steps breakdown of the artifact and its possible uses.

4.4.1 Setup/Jenkins mode

This mode is the initial stage of using the artifact. It takes as its input a ‘snapshot’ .zip archive, which is then peeked in order to identify the necessary debugging information that’s contained within. The software version is identified from files in the snapshot .zip and then the supporting debugging information is extracted from different environments and collaboration tools in the case company servers. After the environment is set up, the files to be debugged are extracted in the root path of the artifact, all possible analysis is run on the extracted files, and the results are then stored in a result folder of the artifact root path.
The numbered list below briefly explains each step of this mode. The same steps can be seen in Figure 3, specifically the Setup and Analysis stages shown. All of the modules illustrated in Figure 3 are described in more detail later in this section.

1. Inspect the snapshot file (supplied as an input parameter).
2. Identify the type of archive file of the snapshot – Format 1.x, 2.x, 3.x (see Section 4.1).
3. Identify the family of the product (family 1, family 2, see Section 4.1 for details).
4. Identify software baseline version from snapshot if present.
5. Prepare the files to be extracted from snapshot for each processor node – multiple nodes possible, each processor node holds a journal file, multiple core files for each processor core or each physical processor, possible extra in-house memory buffers and system memory logs.
6. Download the debug libraries for the identified software version from the case company workflow tools.
7. Extract the prepared files in pre-determined location in the artifact’s root folder.
8. Locate any required extra debug information from the case-company workflow tool (shared libraries and shared debug libraries).
9. Create the local Linux system root environment based on Linux version used in the software version identified (system root environment creation explained in more detail in Section 4.8).
10. Populate the previously created system root environment and add the extra debug information downloaded from the workflow tool, and add other miscellaneous debug information scattered throughout the case-company storage.
11. Run the main analysis loop – analyze journal file, analyze every corefile, and analyze extra in-house debug files.
12. Write the results in a result folder located in the artifact’s root folder (.txt files containing debugger information).
13. Write a context file containing all versioning information identified, as well the path of the created Linux system root debugging folder (the context file is used with GDB and Eclipse modes).

The Jenkins and Setup modes are identical, except for a few minor differences, such as using slightly different native Linux commands that behave differently when using Jenkins. The real difference when using Jenkins is that the workspace is deleted after the artifact is run once. This is because users running the Jenkins job are only interested in the results provided in Step 12 in the numbered list above. The Jenkins mode utilizes a graphical user interface, which displays the .txt files produced by the artifact and allows the users to receive them by e-mail, download them or simply inspect them in the provided Jenkins GUI (Graphical User Interface).

Figure 3, illustrated below, shows a high-level module overview of the setup and analysis parts described in the listed steps above.
Figure 3. Setup and analysis stages high-level module overview
4.4.2 GDB mode

The prerequisite for this mode is the Setup/Jenkins mode to be run once before analyzing the specified corefile.

The numbered list below briefly explains each step of this mode.

1. Read the corefile path passed as an input parameter.
2. Read the context file created during Setup/Jenkins mode.
3. Locate and check Linux the system root environment in Setup/Jenkins mode (path read from context file).
4. Load the key software versions to be used with the GDB process from context file.
5. Launch a GDB process analyzing the corefile supplied during input.

Once the GDB process is launched with all the appropriate shared library paths and supporting debugging information, the command line can be used to use the GDB application as the user sees fit. This allows extensive and custom debugging to be done for problems that cannot be solved using only the information supplied from the Jenkins/Setup mode.

Figure 4, illustrated below, visually shows the high-level component logic of this stage.

4.4.3 Eclipse mode

The prerequisite for this mode is the Setup/Jenkins mode to be run once before analyzing the specified corefile.

The numbered list below briefly explains each step of this mode.

1. Read the corefile path passed as an input parameter.
2. Read the context file created during Setup/Jenkins mode.
3. Locate and check the Linux system root environment in Setup/Jenkins mode (path read from context file).
4. Load key software versions to be used with the GDB process from the context file.
5. Launch the Eclipse CDT binary analyzing the corefile supplied during input.

This mode is basically identical to the GDB mode explained in Section 4.3.2. The main difference is that the Eclipse CDT binary supplies the user with a GUI, which shows all of the debugging information present in GDB in a graphical user interface format. This mode can be useful when a certain problem can be debugged better using more visual information. Figure 4, illustrated above, visually shows the high-level component logic of this stage.
4.5 Environment Control and Environment Parameters

The Environment Control and Environment Parameters modules drive the overall control of the artifact. Environment Control is responsible for executing the appropriate modules for unpacking logs, detecting versions, and specific debugging operations. Environment Parameters is a data only module, which holds all of the input parameters, parameters discovered during analysis, and parameters to do with finalizing the analysis stage.

These two modules act as singletons and interact with every other module in the artifact. The other modules themselves do not interact with each other, but always go through the Environment Control and Environment parameters, in order to obtain the necessary information for their functionality.

The Environment Control and Environment Parameters modules are created after the artifact parses the user input flags (Section 4.6). Their place in the artifact’s design can also be seen in Figure 3 and Figure 4 illustrated above.
4.6 Folder structure

The folder structure is rigid and follows the same structure every time the artifact is run. The files extracted from the ‘snapshot’ can be stored in the Temp folder until they are ready to be processed further, or if they are ready, they are stored in the appropriate Node X folder (numbered folders for each node). The folder structure is illustrated in Figure 5 below.

The Crash folder holds all of the ready to be analyzed files and files to be used in the debugging process. The Node folders each hold their own Linux journal, corefiles and other memory buffers. The Linux system root folder holds all of the shared libraries that the Linux operating system uses to run its native processes. The Downloaded Shared Libraries folder holds all of the shared libraries obtained from the case company workflow tools or other storage.

The Results folder holds all of the .txt files generated by the artifact, which provide debugging information obtained during the analysis process.

The Temp folder does what its name suggests, it is a temporary place to store intermediate files. It is primarily used as a location to store downloaded archives, which need to be extracted in the Crash folder at a later stage in the application’s logic.

Figure 5. Artifact folder structure

Figure 5, illustrated above, outlines the folder structure explained above in this subsection. All of the actual Python code (.py and .pyc files) are stored directly in the artifact’s root folder.
4.7 Input parameters

The main control of logic is done by passing certain input parameters to the instance of the artifact during launch. This process is aided by the ArgParser Python library (Python Software Foundation, 2018a), which provides a framework and basic error checking for passing input parameters to python processes. All of this basic error checking is implemented using groups and rules, which make sure that the combinations of input parameters supplied are valid. This module is the first piece of logic that gets executed when launching the artifact. In essence, this module determines the correct following sequence of required debugging operations.

4.7.1 Input files

There are two types of input that can be analyzed by the artifact – a ‘snapshot’ .zip file or a corefile to be analyzed.

-s (--snapshot) – indicates path of the ‘snapshot’ to be analyzed

-c (--corefile) – indicated the path of the corefile to be analyzed

The -s and -c flags are required and mutually exclusive. If both or none of them are passed to the artifact process during launch, the software will throw the appropriate error exception.

4.7.2 Mode of operation

There are 3 main different modes of operation (explained in Section 4.3). Each mode is selected by passing the appropriate flag when launching the artifact process. The -s flag is associated with the Setup and Jenkins modes and the -c flag is associated with the GDB and Eclipse modes.

-l (--load) - launches Setup mode

-j (--jenkins) – launches Jenkins mode

-g (--gdb) – launches GDB mode

-e (--eclipse) – launches Eclipse mode

The flags -l, -j, -g and -e are required and mutually exclusive. At least one of them must be specified during the program launch.
4.7.3 Type of product family

Due to the impossibility of determining the exact type of product family described in the ‘snapshot’ file, the type of product must be specified during launch. This is achieved by using the -swt (--softwaretype) flag.

For example, when dealing with a ‘snapshot’ from a certain product family, the user would supply:

- swt PRODUCTFAMILYONEKEYWORD (for product family one)
- swt PRODUCTFAMILYTWOKEYWORD (for product family two)

4.7.4 Version overrides

Often, there are atypical software builds or software builds that have missing debugging information on the case company workflow tools. In these cases, there need to be flags that override the software version and/or internal software components versions.

- sw (--softwareversion) – overrides the software version read from the ‘snapshot’
- override1 – overrides the internal software component 1
- override2 – overrides the internal software component 2

-override1 and -override2 have obfuscated names in this paper, due to the proprietary nature of their names.

By using these override flags, the artifact can ignore information obtained from the ‘snapshot’ and use the values supplied by them.

4.7.5 Custom shared libraries override

Often there are atypical software builds or software builds that have missing shared libraries in the case company workflow tools. In these cases, there needs to be a flag that overrides the artifact’s shared library obtaining process. The -cl (--customlibraries) flag supplies such an override, by indicating the path to a .zip file, which contains shared libraries. The .zip pointed to by the -cl flag is then extracted to the appropriate place in the artifact’s folder structure. For example, overriding the shared library process with files located in a .zip file called CustomLibs.zip looks like:

- cl CustomLibs.zip

4.7.6 Extra GDB commands for analysis

The analysis process uses GDB to read each corefile located in the artifact’s folder structure. The commands for analysis are already pre-set and run every time. The flag -cmd (--commandsgdb) gives the option to supply extra commands to be run in the same process. For example, if the user wishes to run the gdb commands ‘bt full’ and ‘info proc mappings’, the flag would be used as such: -cmd ‘bt full,info proc mappings’
4.8 Input format classifier

With the multiple times of input formats possible (See Section 4.1), there is a need of some form of an identifier. The Input Format class does this job by looking at predefined (known) identifiers in the .zip input file. These search operations are hardcoded in the sense that the class searches for certain pre-defined conditions to be met, in order to correctly recognize the type of ‘snapshot’ it is reading.

The Input Classifier class peeks in the .zip file and identifies these key files by going through its pre-defined knowledge list in no particular order. Once a match is found it returns the corresponding Product Family and Format X.X. This functionality relies on the mutual exclusivity of the identifiers for the different input formats. The solution is not the best in terms of long-lasting design, but due to the high variability of the input formats and the inclusion of additional Product Families it is the most cost-efficient at this stage of the artifact.

4.9 Debug libraries module

This module is responsible for collecting all of the base debug libraries associated with a certain build under analysis. Its main task is downloading, collecting and extracting library files scattered throughout the case company’s data storage facilities. There are 3 types of library file locations:

1. The workflow tool associated with the software build.
2. The Linux OS (Operating System) kernel libraries.
3. The specific in-house software component libraries.

The specifics of each of these locations are different depending on the Product Family, and software build and therefore their combinations.

The libraries from type 1 in the list above are downloaded and extracted directly from the case company’s workflow tool. The download links for each library archive is obtained by searching the workflow tool with the version information obtained from reading the ‘snapshot’ input file. The list of download links to be collected is gathered during the reading of the ‘snapshot’. The Debug library module handles the downloading, extraction and placement of the libraries.

The Linux OS kernel libraries from type 2 in the list above are stored in the company’s raw data storage facilities. The libraries themselves technically do not need to be moved and can be read directly from the storage, however the libraries present in that location only hold basic information and lack the additional debug libraries required for extensive debugging. Therefore, the base libraries need to be moved in a location where the artifact has full read and write permission, in order for it to be able to make changes to the structure and add the additional debug libraries. This process is further described in the Shadow System root module in Section 4.9.

The specific in-house libraries from type 3 in the list above are special cases of specific software components. These components create libraries massive in size and they cannot be stored in a raw format normally. The artifact is aware of the location of every module in use and fetches the nested archive holding the specific library. The module then extracts the necessary files from the nested archive and places them in the desired location.
4.10 Custom libraries module

This module is used when the override described in Section 4.5.5 is active. The purpose of this module is to allow for a way to inject freshly generated debugging libraries or libraries which are otherwise unavailable on the data storage facilities of the case company. The custom libraries module completely overrides the Debug libraries module (Section 4.7), in the sense that both of these modules are mutually exclusive. If the user chooses to override the normal libraries procurement procedure, they must supply a full set of the libraries they wish to be used.

The custom libraries are supplied in a .zip format and their path is specified in the argument parser during the input arguments stage. The .zip file is extracted in the artifact’s folder structure analogous to the Debug libraries module (Section 4.7).

4.11 Shadow system root module

This is the module responsible for creating and maintaining a copy of the Linux kernel libraries of the software build under analysis. The underlying product that the artifact is trying to analyze runs on Linux, so the processes involved require the kernel (or root) libraries that are used in the everyday operations of the Linux operating system. The libraries for each specific version of Linux in use by a software build are available in the case company’s storage, however they are write protected. The libraries themselves hold only the basic information required and lack any additional debug libraries, which need to be associated with them. This additional debug information is not stored in the same location and needs to be extracted separately. Due to the write restrictions of the location of the base Linux root libraries a new location, which the artifact needs to have read/write permissions, needs to be created. The main task of this module is to create such an environment and inject the related additional debug libraries.

As the base libraries are only available for reading operations and not writing, there is no need to fully copy the required files. The system root module creates a ‘shadow’ copy of the files by copying the targeted folder structure and creating a symlink (Kerrisk, 2016) for each file present. The ‘shadow’ word in the module’s name is derived from this functionality. The symlink behaves and looks like a normal file, but is minuscule in size compared to the original files.

After the shadow directory of the Linux root is created, the system root module then injects the additional debug libraries by fetching the archives they are contained in and extracting them in the newly created directory.

There are a few special cases of libraries, which are often moved or their generation process is changed regularly. This makes the use of the system root module very convenient, as the artifact can tailor its desired environment without dealing with operating system permissions or fatally changing a location used by other services in the case company.
4.12 Context file

The context file is generated by the artifact after running Setup or Jenkins mode (see Section 4.3.1). The file has a .json format and consists of a dump of the Python dictionary, which holds all of the versioning parameters and folder paths used for debugging. This is implemented by dumping the parameters Python class used to a .json file using the json.dump() command (Python Software Foundation, 2018b).

The context file is read during the GDB and Eclipse processes. It is necessary to ‘teach’ the artifact where the appropriate created folder structures are and what they hold (see Section 4.4). This solves the problem of analyzing the folder structure and trying to gather the pieces every time the artifact is run in GDB or Eclipse mode.

4.13 Analysis modules

The analysis modules work with information extracted from the ‘snapshot’ and download from the case company workflow tools. They use the prepared system root environment (explained in Section 4.8) and other files located in the Crash folder (see folder structure Section 4.4). The analysis is done on the corefiles, the Linux journals and memory buffers identified in the ‘snapshot’. The analysis runs during the Setup/Jenkins mode and write their results in .txt format in the Results folder.

4.13.1 Journal Analyzer

The Journal Analyzer’s role is to parse the Linux journal file, locate exceptions and fatalities, and decode the function addresses found. The Linux journal follows a static structure, which can be relied upon. Each processor on the system can fail and throw an exception, followed by a list of stack traces. The stack traces hold a list of all processes currently running on the particular processes. Each entry in the stack list points to the base address of the function in virtual memory, as well as the offset of the shared library that the function uses. The Journal Analyzer builds a list in memory of all the stack entries with their associated shared libraries and offsets and uses the Linux tool ‘addr2line’ (Free Software Foundation, Inc., 2012) to decode the information in a human readable format. Finally, the Journal Analyzer combines the decoded information with the corresponding raw information obtained directly from the Linux journal and writes it to a .txt file in the Results folder. This whole process is illustrated in the numbered steps list below:

1. Parse the journal file and identify all of the processors that have thrown fatal exceptions.
2. Parse the journal file for each processor found in Step 1 and identify its unique identifier.
3. Parse the journal file for each unique identifier found in Step 2 and create the same number of stack lists.
4. For each stack list, break down into number of processes and obtain their base memory address, library name and library offset.
5. For each broken down stack list in Step 4, decode every entry with ‘addr2line’.
6. Write the decoded information from Step 5 with corresponding information from Step 3.
The end result of this module writes a file for each failed processor. A single processor can fail multiple times. Each file created represents the instance of the failed processor and holds both decoded and raw stack lists for that processor. These analysis files can quickly point to the fault of the crash, due to the fact that the user can now see every failed process, the line number and file name of the offending function.

4.13.2 Memory logs Analyzer

Apart from the corefiles and Linux journal files, there are multiple memory logs present in the ‘snapshot’. The analysis of these memory logs are out of the scope of this paper and require specialized knowledge in order to be analyzed. The case company environment has scripts already prepared to analyze such memory files. The artifact uses those scripts from the case company software collaboration tool to analyze the memory logs and outputs the results in .txt format in the Results folder. This is implemented using the ‘svn extern’ command (Apache Software Foundation, 2017). The artifact checks during run-time whether the scripts were checked out successfully and runs them on the appropriate memory files. These internal tools are named “Internal Analysis Tool 1”, and “Internal Analysis Tool 2” and are illustrated in Figure 3, located in Section 4.3.1.

4.13.3 Corefile Analyzer

The Corefile Analyzer’s purpose is to run a GDB process against every corefile found in the artifact’s root folder structure. In order for GDB to read the corefile correctly, it needs to be pointed to the correct executable files associated with the corefile and the path containing all shared libraries associated with the processes present in the corefile. The artifact opens up a subprocess (Python Software Foundation, 2018b) that launches GDB for each corefile found, which opens a command line GDB process that can be further manipulated by the artifact.

The artifact runs the following GDB commands on every core file:

1. ‘threads apply all bt full’ – This command prints a full backtrace for all processor threads present in corefile (Free Software Foundation, Inc., 2018a).

2. ‘info proc mappings’ – This command prints a memory map of each function and associated library present in the corefile (Free Software Foundation, Inc., 2018a).

3. ‘info shared’ – This command prints a list of shared libraries and their associated debug symbols loaded by GDB (Free Software Foundation, Inc., 2018a).

Each of these commands are run in a separate shell process (Python Software Foundation, 2018) and their output saved in a different file. The user can supply extra GDB commands to be run in the Corefile Analyzer by using the -cmd input flag (see Section 4.5.6).
The numbered steps below give an overview of the whole Corefile Analyzer process:

1. Locate the corefile.
2. Locate the Linux system root folder.
3. Locate the appropriate version of the GDB executable.
4. Open a GDB shell process.
5. Run ‘threads apply all bt full’.
6. Save the shell process output to a .txt file in Results.
7. Open a GDB shell process.
8. Run ‘info proc mappings’.
9. Save the shell process output to a .txt file in Results.
10. Open a GDB shell process.
11. Run ‘info shared’.
12. Save the shell process output to a .txt file in Results.
13. Run any custom GDB commands supplied by -cmd input flag.
14. Save shell process output for each command to a .txt file in Results.
15. Try to locate the next corefile.

The end result of this analysis module is to provide the user with extensive debugging information for each corefile located in the ‘snapshot’, with the possibility of custom user commands supplied by -cmd.

4.14 User Stories

With the design artifact developed as a search process (Guideline 6 Section 2.2.1), user stories fill the role for identifying end requirements extremely well. This subsection will define the user stories with every story being from the point of view of a stakeholder (software developer or tester) using it. The user stories presented only include a high level requirement from the users and do not envelop or describe the underlying process entirely. The user stories were devised by gathering comments, ideas, and desired functionality feedback from stakeholders from within the case company. This process was previously described in Section 4.3 Iterative design. The number of stakeholders involved was not officially recorded, but it included roughly 20 stakeholders.

Tables 2 and 3, shown below, hold all of the user stories collected for the design artifact.
Table 2. User stories defined for the design artifact – part one (of two)

<table>
<thead>
<tr>
<th>ID</th>
<th>User Story</th>
<th>Evaluation Requirement/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>As a software developer, I want to be able to use the artifact to automatically analyze any corefiles on my local system from a supplied ‘snapshot’ input.</td>
<td>Artifact must implement automatic GDB and journal file analysis of corefiles on the user’s system by unpacking the supplied input holding the files.</td>
</tr>
<tr>
<td>2</td>
<td>As a software tester, I want to be able to use the artifact to automatically analyze any corefiles on my local system from a supplied ‘snapshot’ input.</td>
<td>Artifact must implement automatic GDB and journal file analysis of corefiles on the user’s system by unpacking the supplied input holding the files. The process must require little-to-none specialized knowledge from the user.</td>
</tr>
<tr>
<td>3</td>
<td>As a software developer, I want to be able to use the artifact to analyze any corefiles on a Jenkins web service from a supplied ‘snapshot’ input.</td>
<td>Artifact must implement automatic GDB and journal file analysis of corefiles on the case company’s Jenkins system by unpacking the supplied input holding the files.</td>
</tr>
<tr>
<td>4</td>
<td>As a software tester, I want to be able to use the artifact to automatically analyze any corefiles on my local system from a supplied ‘snapshot’ input.</td>
<td>Artifact must implement automatic GDB and journal file analysis of corefiles on the case company’s Jenkins system by unpacking the supplied input holding the files. The process must require little-to-none specialized knowledge from the user.</td>
</tr>
<tr>
<td>5</td>
<td>As a software developer, I want to be able to use the artifact to automatically analyze any corefiles on my local system from a supplied ‘snapshot’ input, while also overriding some of the debugging information with values of my choosing.</td>
<td>Artifact must implement automatic GDB and journal file analysis of corefiles on the user’s system by unpacking the supplied input holding the files. The artifact must allow custom values to be added for the debugging process.</td>
</tr>
<tr>
<td>6</td>
<td>As a software developer, I want to be able to use the artifact to automatically analyze any corefiles on my local system from a supplied ‘snapshot’ input, while also overriding some of the debugging information with values of my choosing.</td>
<td>Artifact must implement automatic GDB and journal file analysis of corefiles on the case company’s Jenkins system by unpacking the supplied input holding the files. The artifact must allow custom values to be added for the debugging process.</td>
</tr>
<tr>
<td>ID</td>
<td>User Story</td>
<td>Evaluation Requirement/s</td>
</tr>
<tr>
<td>----</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>7</td>
<td>As a software developer, I want to be able to use the artifact to start a GDB shell on my local system, in order to analyze any corefiles from a supplied ‘snapshot’ input.</td>
<td>Artifact must implement the initiation of a GDB shell on the user’s system, which can “open” a corefile by unpacking the supplied input holding the files.</td>
</tr>
<tr>
<td>8</td>
<td>As a software developer, I want to be able to use the artifact to start a GDB shell on my local system, in order to analyze any corefiles from a supplied ‘snapshot’ input, while also overriding some of the debugging information with values of my choosing.</td>
<td>Artifact must implement the initiation of a GDB shell on the user’s system, which can “open” a corefile by unpacking the supplied input holding the files. The artifact must allow custom values to be added for the debugging process.</td>
</tr>
<tr>
<td>9</td>
<td>As a software developer, I want to be able to use the artifact to start an Eclipse CDT instance on my local system, in order to analyze any corefiles from a supplied ‘snapshot’ input.</td>
<td>Artifact must implement the initiation of an Eclipse CDT instance on the user’s system, which can “open” a corefile by unpacking the supplied input holding the files.</td>
</tr>
<tr>
<td>10</td>
<td>As a software developer, I want to be able to use the artifact to start an Eclipse CDT instance on my local system, in order to analyze any corefiles from a supplied ‘snapshot’ input, while also overriding some of the debugging information with values of my choosing.</td>
<td>Artifact must implement the initiation of an Eclipse CDT instance on the user’s system, which can “open” a corefile by unpacking the supplied input holding the files. The artifact must allow custom values to be added for the debugging process.</td>
</tr>
</tbody>
</table>

The user stories in the tables above cover all of the possible use cases for the artifact at this stage of development. Its main stakeholders are software developers and software testers. The user stories cover the following “modes” of the application:

- automatic analysis on the local system
- automatic analysis on the local system with overridden values
- automatic analysis using a Jenkins web service
- automatic analysis using a Jenkins web service with overridden values
- starting a GDB shell analyzing a corefile on the local system
- starting a GDB shell analyzing a corefile on the local system with overridden values
- starting an Eclipse CDT instance analyzing a corefile on the local system
- starting an Eclipse CDT instance analyzing a corefile on the local system with overridden values
5. Evaluation

The evaluation section is divided in three parts: time metrics obtained from the artifact compared against the previously existing solution, evaluation based on user stories defined in the Implementation section, and finally comments gathered from semi-structured interviews with stakeholders from the case company.

5.1 Time metrics

This section shows the comparison, in regards to time taken, of the existing manual process solution and the solution that the design artifact implements. All of the results taken and comparisons done are by the author of this research due to time constraints, hence the threat to validity is very high.

The following processes were executed both manually and with the design artifact:

**P1**: Analyze all corefiles present in the ‘snapshot’ without a pre-existing environment

**P2**: Start a GDB shell process opening a corefile without a pre-existing environment

**P3**: Analyze all corefiles present in the ‘snapshot’ with a pre-existing environment

**P4**: Start a GDB shell process opening a corefile with a pre-existing environment

**P1** and **P2** are achieving the same thing as **P3** and **P4**, respectively, with the only difference being their pre-existing environment. The environment in the list above refers to all of the necessary supporting files with which the debugging process can be started successfully. The manual process was done very generously, meaning that there was prior knowledge of the system and where the necessary files are stored. All of the separate operations are also executed quickly and still physically typed, but without taking the time to verify the actual output provided. The timing results of the design artifact iterations were obtained from data collected during its development.

Six iterations with different settings were executed. The main difference between the iterations is the Product Family and the input file format. These properties are previously defined in in Section 4.1. The combinations of the iterations exhaustively cover all currently supported functionality by the design artifact.
Table 5. Time taken for manual execution of P1

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Product Family</th>
<th>Input format</th>
<th>Time taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1.1</td>
<td>17 minutes 23 seconds</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2.1</td>
<td>15 minutes 9 seconds</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3.1</td>
<td>14 minutes 2 seconds</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1.2</td>
<td>25 minutes 30 seconds</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2.2</td>
<td>27 minutes 39 seconds</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3.3</td>
<td>27 minutes 5 seconds</td>
</tr>
</tbody>
</table>

Table 5 holds the results of manually running the P1 process. P1 requires all of the corefiles to be opened and a single command to be run, followed by saving the output printed on the screen to a file or another storage medium. The results are similar to each other and all of them take significant time to achieve. The results for Product Family 2 are higher than Product Family 1 due to the presence of more corefiles in the input ‘snapshot’.

Table 6. Time taken for automatic execution of P1 using the design artifact

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Product Family</th>
<th>Input format</th>
<th>Time taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1.1</td>
<td>4 minutes 4 seconds</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2.1</td>
<td>4 minutes 33 seconds</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3.1</td>
<td>4 minutes 32 seconds</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1.2</td>
<td>5 minutes 11 seconds</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2.2</td>
<td>7 minutes 15 seconds</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3.3</td>
<td>5 minutes 13 seconds</td>
</tr>
</tbody>
</table>

Table 6 shows the results for P1 using the newly crafted automated solution. The time taken is significantly lower than that of P1 executed manually, shown in Table 5. Results for Product Family 2 are higher than Product Family 1 due to the presence of more corefiles in the input ‘snapshot’.
Table 7. Time taken for manual execution of P2

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Product Family</th>
<th>Input format</th>
<th>Time taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1.1</td>
<td>13 minutes 39 seconds</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2.1</td>
<td>12 minutes 25 seconds</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3.1</td>
<td>11 minutes 11 seconds</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1.2</td>
<td>12 minutes 5 seconds</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2.2</td>
<td>11 minutes 9 seconds</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3.3</td>
<td>12 minutes 23 seconds</td>
</tr>
</tbody>
</table>

Table 7 holds the results of manually running the P2 process. P2 requires only a single corefile to be ‘opened’ with GDB. The time taken to set up the environment was similar to the results for manual execution of P1, but the time taken is lower due to the requirement of ‘opening’ only a single corefile. Most of the necessary operations are for setting up the environment required to be able to run the debugging process.

Table 8 shows the results for P2 using the newly crafted automated solution. As noted for P1, the time difference between the manual and automated solution is huge. This is due to the time required to set up the environment, which is sped up by using the automated solution. The difference between Product Family 1 and Product Family 2 is smaller, due to the requirement of only ‘opening’ a single corefile from the input ‘snapshot’.

Table 8. Time taken for automatic execution of P2 using the design artifact

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Product Family</th>
<th>Input format</th>
<th>Time taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1.1</td>
<td>3 minutes 50 seconds</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2.1</td>
<td>4 minutes 15 seconds</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3.1</td>
<td>4 minutes 20 seconds</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1.2</td>
<td>4 minutes 30 seconds</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2.2</td>
<td>4 minutes 53 seconds</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3.3</td>
<td>4 minutes 23 seconds</td>
</tr>
</tbody>
</table>
Table 9. Time taken for manual execution of P3

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Product Family</th>
<th>Input format</th>
<th>Time taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1.1</td>
<td>1 minute 35 seconds</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2.1</td>
<td>1 minute 12 seconds</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3.1</td>
<td>1 minute 34 seconds</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1.2</td>
<td>3 minutes 47 seconds</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2.2</td>
<td>2 minutes 57 seconds</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3.3</td>
<td>2 minutes 45 seconds</td>
</tr>
</tbody>
</table>

Table 9 shows the results manually executing P3. P3 is ‘opening’ all of the corefiles with a pre-existing environment in place. With the environment present, the process to analyze the corefiles is straightforward and requires few commands as input.

Table 10. Time taken for automatic execution of P3 using the design artifact

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Product Family</th>
<th>Input format</th>
<th>Time taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1.1</td>
<td>45 seconds</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2.1</td>
<td>59 seconds</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3.1</td>
<td>53 seconds</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1.2</td>
<td>1 minute 32 seconds</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2.2</td>
<td>1 minute 6 seconds</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3.3</td>
<td>1 minute 45 seconds</td>
</tr>
</tbody>
</table>

Table 10 shows the results for P3 using the newly crafted automated solution. The difference between the automated solution and the manual process for P3 is much smaller here. This is due to the brunt of the work falling on setting up the environment.
Table 11. Time taken for manual execution of P4

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Product Family</th>
<th>Input format</th>
<th>Time taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1.1</td>
<td>12 seconds</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2.1</td>
<td>11 seconds</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3.1</td>
<td>9 seconds</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1.2</td>
<td>9 seconds</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2.2</td>
<td>10 seconds</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3.3</td>
<td>9 seconds</td>
</tr>
</tbody>
</table>

Table 11 shows the results manually executing P4. P4 is only ‘opening’ a single corefile and the environment is already set up. The really low time taken is because only a single command needs to be typed.

Table 12. Time taken for automatic execution of P4 using the design artifact

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Product Family</th>
<th>Input format</th>
<th>Time taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1.1</td>
<td>15 seconds</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2.1</td>
<td>16 seconds</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3.1</td>
<td>20 seconds</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1.2</td>
<td>17 seconds</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2.2</td>
<td>19 seconds</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>3.3</td>
<td>15 seconds</td>
</tr>
</tbody>
</table>

Table 12 shows the results for P4 using the newly crafted automated solution. The time taken is similar to the manual execution of P4, but slightly higher due to the time taken to run the whole application code of the artifact.

5.1.1 Summary of time metrics

The time taken by the design artifact is significantly lower than the manual solution. The huge difference is due to the time required to set up the environment holding all of the libraries and necessary supporting files. These are input/output operations, which take a long time to type and can be very susceptible to typos. Given that the manual execution was done in a highly optimistic manner, the same operations would take much longer to be completed by a person who is entirely unfamiliar with the debugging process itself and is unaware of the locations of all the required files. The only case where the manual
execution of the debugging process is faster, is when the environment is already set up and the desired debugging command is simple.

5.2 User stories

The artifact is also evaluated against the user stories defined in Section 4.13. The success is indicated by marking user stories with color (Green/Red) and status (Yes/No) of completion.

Table 13. User stories status of completion

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Status - Yes</td>
<td>Status - Yes</td>
<td>Status - Yes</td>
<td>Status - Yes</td>
<td>Status – Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Status - Yes</td>
<td>Status - Yes</td>
<td>Status - Yes</td>
<td>Status - Yes</td>
<td>Status – Yes</td>
</tr>
</tbody>
</table>

In the end, all of the pre-defined user stories were completed successfully, as the specified requirements and functionality, from the point of view of the different stakeholders, were met. Table 13 shows the status of each user story.

The results were confirmed in an unofficial setting, by verbal and visual confirmation by the involved stakeholders.

5.3 Semi-structured interview comments

In order to further evaluate the artifact, qualitative data was gathered in the form of semi-structured interviews. This kind of evaluation opens up result gathering regarding the human factor, which is often neglected.

5.3.1 Interview environment and interviewee information

The interviews were conducted at the case company’s offices during working hours in a casual setting. The semi-structured format was chosen, due to its greater flexibility and lesser impact on the working time of the interviewees. All of the interviews were conducted face-to-face with only the interviewer and interviewee present. Every interview was recorded, and later transcribed, with the consent of the interviewee.

Precautions were taken in order not to intrude on the typical social setting already present in the working environment. All of the interviewees were familiar with the interviewer and were confident that their confidentiality would be strictly kept. The interviews were conducted at only one of the case company’s sites due to time restrictions and logistics issues.

Each session took roughly 20 minutes at a time convenient for the interviewees. The questions and structure followed roughly followed the form located in the Appendix at the end of this paper.
Overall, there were 5 interviews conducted with 5 stakeholders from different backgrounds. Table 14, below, indicates the interviewees’ job titles and responsibilities.

**Table 14. Classification of interviewees by job position and responsibilities**

<table>
<thead>
<tr>
<th>Group</th>
<th>Interviewees (numbered)</th>
<th>Qualifications/Job responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Test Engineers</td>
<td>STE 1</td>
<td>Responsible for creating, running and analyzing adequate test cases for the case company’s product. Also responsible for managing the test environment.</td>
</tr>
<tr>
<td>Software Component Engineers</td>
<td>SCE 1, SCE 2</td>
<td>Responsible for developing business logic code regarding the case company’s business.</td>
</tr>
<tr>
<td>Platform Software Engineers</td>
<td>PSE 1, PSE 2</td>
<td>Responsible for developing the underlying platform on which the case company’s product runs on.</td>
</tr>
</tbody>
</table>

The interviewees were chosen to be from mixed backgrounds to minimize the so-called *elite bias* (Myers & Newmann, 2007). Having interviewees from mixed backgrounds lowers the risk of encountering any focused bias on obtaining feedback from favorable to the research topic individuals.

The interviewees were asked the pre-set questions and were encouraged to further discuss their thoughts by the interviewer. Each question roughly set the topic of discussion and the interviewer guided the interviewees through the topics that occurred naturally during conversation. In some instances, the structure shown in Appendix A did not need to be followed at all, due to the discussion leading in all of the areas targeted by the pre-set questions.

### 5.3.2 Limitations of the interviews

The set of interviews is only 5, which is too low of a number in order to perform any type of statistically significant analysis. All of the interviewees were already familiar with the artifact under development and may or may not be biased towards it, as they already possessed some previously obtained knowledge.

The questions were constructed, in order to maximize unbiased answers, but there is always the risk of the *Hawthorne effect* being present (McCambridge, Witton & Elbourne, 2014). The result of the interview could be biased simply by the interviewer being present in the room.
The interviews were conducted in English, which is not the mother tongue of the interviewer or any of the interviewees, which could have possibly led to eventual misunderstanding of the topics discussed.

Another limitation would be the lack of software testers interviewed. This is a key target group, which would be able to benefit the most from the artifact’s learning process possibilities.

### 5.3.3 Interview results

The content analysis of the interviews will be done conventionally. Meaning, that the questions asked are open-ended and unbiased in order to limit the influence on the interviewee’s opinion on the topic and lessen any pre-categorization of his/her opinions. The method of conventional analysis is most useful and most used when the existing theories and scientific literature on the topic are limited (Hsieh & Shannon, 2005). The interviewer’s role was to engage the interviewee through the occurring discussion regarding the open-ended questions and capture the interviewee’s first impressions, thoughts, and analysis. The interviewees’ roles are defined in Table 5 located in Section 5.3.1.

Tables 15, 16, 17, 18, and 19 below summarize the general opinion of the interviewee for each respective question. Each of the tables is followed by a short summary of the general sentiment of the interviewees.

**Table 15. Interviewee sentiments for topic 1 of the interview (questions in Appendix A)**

<table>
<thead>
<tr>
<th><strong>Question 1: General topic – How often do you spend time looking at problems related to debugging Linux corefiles?</strong></th>
<th><strong>Interviewee comments</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Interviewee</td>
<td>Interviewee comments</td>
</tr>
<tr>
<td>STE 1</td>
<td>Interviewee’s position is not very related to the debugging of post-mortem files – needs to analyze a corefile maybe once a month. Only some of his/her work is relevant to the automated tool.</td>
</tr>
<tr>
<td>SCE 1</td>
<td>The interviewee needs to analyze post-mortem debugging problems roughly once a week.</td>
</tr>
<tr>
<td>SCE 2</td>
<td>The interviewee indicated that post-mortem debugging processes take up roughly 60% of his working time.</td>
</tr>
<tr>
<td>PSE 1</td>
<td>The interviewee needs to analyze post-mortem debugging problems. The interviewee personally needs to deal with post-mortem debugging problems once a month. However, he/she indicated that a few people on his team need to deal with such problems daily.</td>
</tr>
<tr>
<td>PSE 2</td>
<td>The interviewee needs to deal with post-mortem debugging processes roughly twice a week.</td>
</tr>
</tbody>
</table>
The people interviewed do not need to deal with post-mortem debugging daily, but some very often. However, some of the interviewees have indicated that there are a lot of people that face post-mortem debugging problems several times a day.

Table 16. Interviewee sentiments for topic 2 of the interview (questions located in Appendix A)

<table>
<thead>
<tr>
<th>Interviewee</th>
<th>Interviewee comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STE 1</strong></td>
<td>Eliminates the need of studying where all specific files are located, such as libraries and other miscellaneous items. It makes debugging faster and helps the person concentrate on the actual problem, rather than the details.</td>
</tr>
<tr>
<td><strong>SCE 1</strong></td>
<td>The artifact drastically lowers the time in which a post-mortem debugging problem is faced. The process of post-mortem debugging is currently very complicated, so it is extremely problematic for newcomers.</td>
</tr>
<tr>
<td><strong>SCE 2</strong></td>
<td>The artifact speeds up the debugging process a lot and makes it easier to start facing the problem. The threshold to start the investigation is much lower.</td>
</tr>
<tr>
<td><strong>PSE 1</strong></td>
<td>It really cuts down on the repetitive tasks that need to be done every single time facing a post-mortem debugging problem. With the scale of the case company, such an automated process is very useful and cuts off a lot of time required to run the process. The tool would help by focusing on identifying commonly occurring problems.</td>
</tr>
<tr>
<td><strong>PSE 2</strong></td>
<td>The artifact cuts down on the time required to run the debugging process immensely and would save the engineers looking at such problems a lot of time. This would especially help people that do not need to deal with such problems very often.</td>
</tr>
</tbody>
</table>

The general opinion of the interviewees about the possible benefits of the artifact is extremely positive. All of them agree that the automated solution would drastically decrease the time necessary to analyze post-mortem debugging problems.
Table 17. Interviewee sentiments for topic 3 of the interview (questions located in Appendix A)

<table>
<thead>
<tr>
<th>Interviewee number</th>
<th>Interviewee comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>STE 1</td>
<td>It helps new people, because they do not need to worry about the specifics and can just use the tool or follow the process done by the tool initially.</td>
</tr>
<tr>
<td>SCE 1</td>
<td>It will drastically decrease the learning time necessary for freshly hired people and non-technical staff in general.</td>
</tr>
<tr>
<td>SCE 2</td>
<td>The interviewee agrees that the artifact will help new people learn the process. The users could rely on a good first analysis performed by the artifact initially and they could just follow the same process without having to worry about tool or software architecture related problems.</td>
</tr>
<tr>
<td>PSE 1</td>
<td>It will definitely allow new people and non-technical staff to gain more insight about the process and could potentially serve as a tool focused completely on learning the process. The interviewee emphasized that this design artifact would help testing staff immensely.</td>
</tr>
<tr>
<td>PSE 2</td>
<td>It will definitely allow new people and non-technical staff to learn the process very quickly, especially people that are not related to software engineering, such as testers.</td>
</tr>
</tbody>
</table>

The interviewees unanimously think that the design artifact would be a great tool to use in order to teach new hires or non-technical staff about the post-mortem debugging process. It could help people that are not familiar with a specific process or people that have no knowledge of it at all.
Table 18. Interviewee sentiments for topic 4 of the interview (questions located in Appendix A)

<table>
<thead>
<tr>
<th>Interviewee</th>
<th>Interviewee comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>STE 1</td>
<td>The tool could be hiding details or it could fail silently, which could cause the engineer having to debug the actual artifact first, rather than the particular debugging problem he is facing. Big drawback identified is the need for maintenance. The artifact needs to have a clearly selected person in charge of the maintenance.</td>
</tr>
<tr>
<td>SCE 1</td>
<td>A lot of the software components in the case company have a lot of differences, so it could cause a lot of maintenance issues.</td>
</tr>
<tr>
<td>SCE 2</td>
<td>Users of the design artifact might get used to relying on its results and might miss an error they could otherwise find if they go through the process themselves.</td>
</tr>
<tr>
<td>PSE 1</td>
<td>The automation of the process would cause people to become complacent and rely on it too much. Interviewee compared it to using a calculator when the user is not familiar with all of the arithmetic operations. The interviewee also mentioned that people are usually resistant to change and the artifact must really flaunt its possible benefits, so people would be more willing to use it.</td>
</tr>
<tr>
<td>PSE 2</td>
<td>The interviewee identified the need of maintenance as the artifact’s biggest drawback.</td>
</tr>
</tbody>
</table>

The general drawback identified is the need for maintenance. Another flaw pointed out is that the users of the design artifact might come to rely on it too much and potentially miss fatal errors, which the artifact did not discover.
Table 19. Interviewee sentiments for topic 5 of the interview (questions located in Appendix A)

<table>
<thead>
<tr>
<th>Interviewee number</th>
<th>Interviewee comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>STE 1</td>
<td>The interviewee would use the tool every single time if it is applicable.</td>
</tr>
<tr>
<td>SCE 1</td>
<td>The interviewee would use the tool every single time if it is applicable.</td>
</tr>
<tr>
<td>SCE 2</td>
<td>The interviewee indicated that the first thing he would do, after receiving a post-mortem debugging problem, is try the artifact.</td>
</tr>
<tr>
<td>PSE 1</td>
<td>The interviewee would use the tool every single time if it is applicable. The interviewee mentioned that the process of obtaining all of the required information for the debugging process is extremely onerous.</td>
</tr>
<tr>
<td>PSE 2</td>
<td>The interviewee would use the tool every single time if it is applicable.</td>
</tr>
</tbody>
</table>

All of the interviewees demonstrated great interest in using the design artifact when they are facing problems with which it can help.

5.3.4 Summary of interview results

The feedback from the interviewees was generally positive. Negative points identified have to do with the maintenance of the tool, which ultimately relies on every other process in the case company regarding the software development and maintenance of other case company components. Another drawback identified is the possibility where users of the artifact would come to rely on it too much, which could lead to inability to operate normally if the artifact is not operational or the users might get used to the artifact doing all of the work for them and might miss problems, which the artifact cannot find or has missed. However, the benefits far outweigh the drawbacks. The interviewees were confident that the design artifact would drastically lower the time required to analyze post-mortem debugging process and would present a great learning opportunity for new hires and non-technical staff. The interviewees were keen to use the artifact and would use it every single time they face an applicable problem.
6. Discussion and design science guidelines evaluation

The objective of this research was to provide a solution for automating Linux post-mortem debugging, which would improve the current solutions and workings available. In order to be able to judge the effectiveness of the new solution, it was compared to what was set in place before it in the environment under research. The design science artifact was implemented successfully and used by several stakeholders from the research company. The results obtained were in the form of time metrics taken from the design artifact compared against the previous existing solution and from interview comments from the same stakeholders of the company. The data collection and results are shown in Section 5, called Evaluation. This section will discuss the results by summarizing them, pointing out interesting findings and trying to answer the research questions posed in Section 2.1. Later on, the conducted research will be evaluated against the design science research guidelines defined in Section 2.2.1.

Overall, the response was extremely positive. The stakeholders were confident in the benefits of the tool and showed huge interest in using it in the future when applicable. When using the design artifact, the time savings were significant. In some cases, there were 20 minutes shaved off of the time taken. These results are really promising, because the design artifact itself is not mature yet. With proper maintenance, support, and development, there is a high possibility that the underlying process will be greatly improved. The main benefit identified of the design artifact is its ability to quickly configure the required environment for the Linux post-mortem debugging process. The task of creating the environment is the most repetitive one and the one that requires the most insider knowledge, meaning knowing the location of certain files and libraries on the case company’s system. With the tool supporting the creation of such an environment, it was really easy for people to follow the process and greatly learn from it.

The main drawback identified was the need of constant maintenance. This is due to the instability and high variability of the underlying processes that the design artifact solution relies upon. The processes required are different for every processor family used in the company, and the variability combinations grow even higher when you start analyzing separate software components. It is sensible that a person or a team must be responsible for maintaining the solution, in order for it to be correctly operational and reliable. It was noted by some of the people interviewed for this paper, that some stakeholders might grow too used to using the tool and might ignore bad results or lack of results.

The related work review conducted in Section 2 Related work identified trends and benefits regarding the benefits of automating debugging processes and the problems that specialized processor constructs potentially involve. The design artifact solution focuses mostly on the utilization of already available information, which can be collected and implemented easily (Huselius et al., July 2003). Another confirmed finding was the fact that embedded systems solutions often have long up-times and the size of the log information can be humongous (Huselius et al., July 2003). The great size of such files makes the manual execution of these debugging processes cumbersome for the users, as the commands can be extremely error-prone and the processes themselves are highly variable. Also, the related work section identified the benefit of tool reusability for specific environments (Myers et al., 2011). This can be highly capitalized upon in the case of the case company’s environment, due to its large size.
The crash analysis, data representation, and automated error reporting is a large field of study (Murphy, 2004). Often, the automated post-mortem debugging process is implemented both on the client and server side, especially when dealing with large-scale systems (Bligh, Desnoyers & Schultz, 2007 June). Meaning, the automated pipeline touches the whole application lifecycle and not just one side of it (Glerum et al., 2009 October) (Hilgers, Macht, Muller & Spreitzenbarth, 2014 May). In most cases, the solutions offer further analysis of the extracted data, but they ultimately find raw data extracted more useful than processed or visualized data (Krantz, 2015) (Auguston, Jeffery & Underwood, 2003). Despite best efforts, the related work part of this study could not identify any scientific work that is trying to assemble an artifact with extremely similar functionality and end goal. The identified previous work in the field either uses automated post-mortem debugging at an extremely large scale and/or it is integrated directly into the continuous integration pipeline. The main difference in this research, is the entire focus of the artifact to maximize the time-saving benefits of automating the post-mortem debugging process. The other difference is the way that the software engineering pipeline is currently implemented in the case company. The process is not mature at all and the input files have no centralized repository. The input files are often sent by e-mail or kept in arbitrary storage locations with no established specification. At this point, the design artifact cannot be integrated in the case company’s other automated reporting processes. In order to directly compare the design artifact, created during this research, improvements to the case company’s software ecosystem need to be made.

The first research question posed was: How much time does an automated solution save, as opposed to doing the process manually?

The time saved can be found in Section 5.1. The savings were significant, especially this stage of maturity of the design artifact. The process of measuring the time taken of the previous existing solution is not perfect, therefore it was decided to be as optimistic as possible. Even when placing such a handicap, the results were more than positive. Potentially, the design artifact could save at least 10-15 minutes of every single problem involving Linux post-mortem debugging processes. Furthermore, it could allow people that have no knowledge of the debugging process and how it is normally conducted, to learn by themselves, potentially saving even more time for the engineers, who would normally have to deal with this. Overall, the potential of time savings is big and very promising.

The second research question posed was: How does the amount of pre-knowledge required differ from the automated solution, as opposed to the manual one?

As mentioned in the short discussion above, the identified learning potential of the design artifact is great. Even with the current solution in place, it could serve as a learning tool for new hires and non-technical staff. In the future, if this is the most desired trait of the artifact, it could be easily modified to serve that purpose better. However, this does not mean the artifact has to be entirely focused on learning. It could serve both as a good debugging tool and as a learning opportunity, possibly both at the same time. Overall, the artifact’s potential as a learning tool is big, as mentioned by the interview comments collected in this paper.

The main research question posed was: How can an automated solution for Linux post-mortem debugging be implemented?

The specific implementation details are located in Section 4 Implementation. It can be seen that the artifact requires a lot of file handling operations and previously specified
key locations of required for the debugging process files. The functionality itself is not complicated at all, however, the number of file operations is extremely high. The previous statement, coupled with the high variability of these processes in the case company, highlight the fact that such solutions are very hard to generalize, as the most significant factor is the environment in which they are being built. Overall, the design process followed the previously existing manual solution. There were no special tools or processes required during the development of the design artifact. Such a solution can be built for any Linux system that has all of the necessary shared libraries, debug information, and log information available by using commonly available open-source tools and software.

6.1 Evaluation of design science research guidelines

The design science guidelines were previously specified in Section 2.2.1 This subsection will try to evaluate them based on the current implementation of the design science artifact.

Guideline 1: Design as an artifact

The artifact was designed as a functioning automated post-mortem debugging tool. It provides useful automatically extracted debugging information, as it was posed initially.

Guideline 2: Problem Relevance

The scope was kept to the problem relevance. The business value was proved in Section 5 Evaluation. Also, the potential of the artifact as a learning tool for non-technical was proved by the interview comments gathered.

Guideline 3: Design Evaluation

The design evaluation, as mentioned before, had no clear universally defined metric. The utility of the artifact was demonstrated by time metrics collected comparing the previously existing manual solution to the newly designed artifact solution. The evaluation also included feedback gathered from semi-structured interviews with stakeholders.

Guideline 4: Research Contributions

The research contribution was the artifact itself. It solved the posed problem in an efficient manner and could be used to push the existing scientific knowledge in a business environment. The related work part of this research could not find any scientific work involving similar solutions.
Guideline 5: Research Rigor

In this case, research rigor refers to the way in which the research is conducted. As the environment in which this research is conducted cannot be very formalized, the rigor will be assessed with respect to its applicability and generalizability.

The applicability and generalizability of the artifact were previously discussed in this section. Its applicability was proven to be a success. The generalizability aspect is difficult to prove, as the requirements for implementing such an artifact for different systems heavily depends on the underlying environment in which it will be built.

Guideline 6: Design as a search process

The search process was mainly involved in the implementation stage of this research. The modularized architecture of the artifact proved to be an efficient way of building it. That way, every module could be further optimized in its own scope, without affecting the rest of the functionality. With the modularized architecture, further features can be easily added without impacting the existing functionality. In a way, the artifact was implemented with great flexibility so it could always be tailored for its environment, even if it changes greatly.

Guideline 7: Communication of Research

The research was mainly aimed at a technically-oriented audience, but is sufficiently accessible for laymen in the field. The text explains every feature described in an abstracted manner, making it easier to understand for people unfamiliar with the subject matter.
7. Conclusion

The objective of this research was to provide a solution for automating Linux post-mortem debugging, which is better than the current solutions and workings available. The research was conducted at a large-scale telecommunications company, which did not have an automated solution available. The effectiveness of the new solution was judged by comparing it to user stories gathered from the stakeholders, time metrics comparing the previous and newly created process, and by gathering comments from semi-structured interviews with the stakeholders. In the end, the effectiveness of the tool was proven, in regards to its time saving capabilities, as well as its potential as a learning tool, which can be used to teach new staff or non-technical staff. The major drawbacks identified were the need of maintenance, and the possibility of the stakeholders getting too used to the artifact and potentially missing critical failures or depending on its output too much.

The design artifact was distributed to certain stakeholders within the case company and the initial results were very promising. Even when the current development state of the artifact is not very mature, the benefits, and potential future use cases identified are numerous. The solution implemented is generic enough, so that it can be built for any Linux system, which has made all of the necessary files for the debugging process available.

7.1 Limitations

Due to the way the case company’s tool system is implemented, the design artifact cannot be currently integrated into the automated continuous integration environment of the case company. This makes the comparison of the design artifact created by this research to other prior scientific work difficult.

The data gathered in the evaluation stage of this research is very limited and collected in a mostly unofficial setting due to time constraints. In order to be able to extract any valuable concrete conclusions backed by the data specifically, the data gathering process must be improved and repeated.

Ultimately, this type of design artifact is highly dependent on the environment in which it is being implemented. Therefore, there can be cases where such a solution is completely inapplicable. For example, if the variability is so great that changes need to be made every single day, or if it is too costly for the build system to store certain files required for the post-mortem debugging. In such cases, new ways of approaching the problem must be explored.

7.2 Future work

The artifact is currently available for only a small subset of the potential stakeholders within the case company. Once the development reaches a mature state, it can be propagated throughout the whole organization, which will make the collection of future results and feedback much easier. With more users, the data gathered will have larger statistical significance, which will open more options for future research work involving the artifact and its implications from many facets.

The research managed to identify that the most time savings came from the automated creation of the debugging environment. This could be delved into deeper, in order to
pinpoint the exact operations, which take up the most time taken to execute the artifact. If those operations are few, they could be improved upon on a different organizational level within the case company and potentially improve its whole software engineering pipeline.

The time metrics obtained and the comparisons made in the evaluation part of this research were done by the author, therefore the validity of the metrics is questionable. In order to obtain data, which would truly show the difference between the manual and automated process, subject groups need to be vetted and the measurements must be taken in a controlled manner. The subject groups should include users with varying knowledge of the system, from completely unfamiliar to comfortable with using it.

The automation of post-mortem debugging could potentially be fully integrated into the case company’s software engineering pipeline. Currently, this is not feasible due to the way the environment in the case company is set up. Once the system is mature enough for this to be possible, the benefits of the automation can be explored further and a direct comparison to other similar solutions can be made.
References


Appendix A. Semi-structured interview form

<table>
<thead>
<tr>
<th>SEMI-STRUCTURED INTERVIEW FORM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Questions</strong></td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>1. How often do you spend time looking at problems related to debugging Linux corefiles?</td>
</tr>
<tr>
<td>2. How would using the developed automated tool help you?</td>
</tr>
<tr>
<td>3. Would the automated tool help people learn the debugging process?</td>
</tr>
<tr>
<td>4. Do you think there are any drawbacks with the using the automated debugging tool?</td>
</tr>
<tr>
<td>5. How often do you think you would use the tool/artifact when facing such debugging problems?</td>
</tr>
</tbody>
</table>