Norwegian School of Economics

Bergen, autumn, 2014

The Exercise of Covenant Defeasace Options

"A study of the removal of restictive covenants in US corporate bonds, by means of big-data analysis"

Nils Diderik Grøttheim Algaard & Jo Magnus Tenfjord

Supervisor: Associate Professor Carsten Gero Bienz

Master thesis in Financial Economics

NORWEGIAN SCHOOL OF ECONOMICS

Abstract

This thesis aims to examine the exercise of covenant defeasance options. To find what bonds are defeased, we build a SEC crawler to analyze more than 1.4m SEC filings. Our methods of analysis are descriptive statistics and regression analysis. The regression analysis is performed by joining our data with Mergent's Fixed Income Securities Database (FISD). Our major findings are: (1) 0.56% of defeasible bonds have this option exercised; (2) defeasance and repurchase are linked together as firms often repurchase as many bonds as possible while any hold outs are removed via defeasance; (3) no evidence that defeasance exercise is clustered in industries with higher uncertainty (4) bonds with a higher number of restrictive covenants are more likely to exercise their defeasance option; (5) there is no indication that callable bonds substitute for defeasance exercise; (6) the defeasance exercises are often linked to major corporate actions, such as acquisitions, mergers or refinancing.

Table of Contents

Preface

This thesis concludes our independent study work in our master's degrees in Financial Economics at the Norwegian School of Economics (NHH).

Our motivation for the choice of thesis subject was multifaceted. The fixed income financial asset class is a large and important part of the financial system. The opportunity to gain deeper insight into this interesting topic is in our opinion useful knowledge for the future. In addition, our specific topic has been subject to limited research. One consequence of this is that there are less academic sources on the subject, but it also meant that we had the opportunity to gain unique insights in the subject and contribute to the understanding of the specifics about covenant defeasance exercise. To the best of our knowledge, there existed no previous data on the subject. The possibility to leverage the use of available comprehensive databases and self-developed software to compile a unique dataset was considered an exciting challenge. In addition, designing the self-developed software in a manner that made it possible to use in other research was rewarding.

Our supervisor, Associate Professor Carsten Gero Bienz, is one of the authors of the most recent research paper on covenant defeasance options. This means that we had access to a leading professional on the subject. We would like to thank him for guidance, constructive feedback and support during our work.

Bergen, December 2014

Nils Diderik Grøttheim Algaard and Jo Magnus Tenfjord

1 Introduction

Bond issuing firms are sometimes presented with situations where value increasing actions are blocked by restrictive bond covenants. The firms are thereby incentivized to renegotiate or circumvent these covenants. Renegotiations of debt contracts are quite common, as Sufi and Roberts (2009) find that 90% of all bank loans are renegotiated to some extent over their maturity period. However, when it comes to bond issues, renegotiation is more complicated as bond issue ownership is spread across many investors. According to Bradley and Roberts (2004), renegotiation is virtually impossible, as the Trust Indenture Act of 1939 states that a two-third approval from external bondholders is necessary to remove covenants. One way covenants can be removed is that the issuer repurchases the outstanding debt.

Brandon (2013) finds in his research paper that "[…] firms are more likely to repurchase outstanding debt either by open market transactions or tender offers when investment frictions are relatively high." One way to do this is to issue callable debt, which can be bought back at a pre-specified price level. Such an option comes at a cost to the issuer. In addition to the repurchase premium above the market price of the bond ex post, there is also a yield premium, which compensates the borrower for refinancing risks. Whether or not a call option is added in a bond issue is therefore a trade-off between flexibility and cost.

Kahan and Rock (2009) show how activist bondholders can pursue unenforced breaches of covenants. These bond investors seek to gain on unenforced covenants by either forcing a default of the bond, or threatening with default to achieve higher repurchase price.

One way to remedy this is the inclusion of a covenant defeasance (or "Legal Defeasance") option. This option is granted to the bond issuer and gives them the right to remove covenants by guaranteeing bond payments by depositing cash or other risk free securities in a restricted escrow account. By doing this, the bondholders continue to receive their coupons and face value at schedule and the bond issuer is released

from the covenants associated with the bond (Mergent, 2014). Initially, this option may seem similar to a call option, but there are distinct differences. As the defeased bond does not trigger any transaction for the bondholder, and thereby no gain or loss, defeasance does not trigger any taxation. In addition, there is no reinvestment risk since the payments of the original bond continues according to the initial schedule. Bienz, Faure-Grimaud and Fluck (2013) show that defeasance is a mechanism that allows to pre-package bond covenant renegotiation. They find that the inclusion of a covenant defeasance option increases the chance of more covenants in a bond issue and because of this, the bond issues command a lower yield and thereby lower capital costs for the firms. Bonds with a covenant defeasance option thereby have a cost advantage in comparison to callable bonds.

Bienz et al (2013) do not look at defeasance exercise, but focus on the inclusion of defeasance indenture agreements. We want to explore the exercises of covenant defeasance and examine when and why corporate bonds are defeased.

This is not a trivial question, as up to date there exists no comprehensive dataset on the exercise of defeasance options. We use a self-developed search program to crawl the Securities and Exchange Commission's (SEC) database (EDGAR) and examine more than 1.4 million US company filings to create a comprehensive dataset on covenant defeasance exercise.

By linking our findings with Mergent's Fixed Income Security Database (FISD), we are able to compare our findings of covenant defeasance exercise with other US corporate bonds.

In our total sample, we find 40 occurrences of covenant defeasance exercise in the US corporate bond market. FISD reports that 7190 bonds have been issued with a defeasance option, which gives a covenant defeasance percentage of 0.56%. This can be regarded as low compared to the 12.07% of bonds that have repurchase offers made in the FISD database.

When performing bond repurchases, bondholders may choose to refuse repurchase offers. This can be to obtain higher repurchase prices due to hold-up as suggested by Bienz et al (2013) or to force a default of a security due to breach of covenant terms as suggested by Kahan and Rock (2009). A possible reason to exercise covenant defeasance options may therefore be to remove any remaining bondholders after repurchase.

Our findings show that there is indeed a link between the tender offers and covenant defeasance. We find that 72.5% of the bonds had previous exchange or tender offers before they were defeased. Of the defeased bonds that were tendered, the tendering was largely successful as the majority of the tendered bonds had tender acceptance rates above 90%. Half of the tendered issues had acceptance ratios above 80%. Regression outputs indicate that bonds that are exchanged or tendered are more likely to have had their defeasance option exercised.

It is possible that some industries have business traits that lead to increased use of covenant defeasance exercise. Bienz et al (2013) show that financially constrained firms with high growth opportunities and higher degree of uncertainty are more likely to include the defeasance option. One example could be the pharmaceutical industry, where companies develop drugs under tight financial constraints. Due to high uncertainty, they are forced to accept restrictive covenants in order to secure financing. Should they get a patent and an FDA approval for a new drug, the uncertainty is significantly reduced, and the need for financing to put the drug to market is increased. By exercising their defeasance option, they can remove restrictive covenants, get better financing, and incur additional debt.

When examining the industries of the defeased bonds, we found that defeasance exercise is distributed to a wide variety of industries. There might be indications that companies in the casinos and gaming industry are more likely to exercise defeasance options than other industries, but this cannot be conclusively decided. Legg and Tang

(2010) show that the casinos and gaming industry was seen as less exposed to systematic risk in the period the covenant defeasance exercises were observed. It is therefore difficult to characterize the casinos and gaming industry as having especially high uncertainty.

One might argue that any method of removing bond covenants is a potential substitute for covenant defeasance. We therefore wish to investigate if call options substitute for defeasance options to remove covenants. Unlike tender or exchange offers, the call option is exercised by the bond issuer. The bondholder cannot refuse the exercise of the call option. This potentially limits activist activity from bondholders.

Opposing this view, Bienz et al (2013) point out that a large number of the callable bonds are issued at make-whole premium. Half of the bonds that carry both a call and defeasance option have to be called at a make whole premium. A make whole premium comprises the net present value of all outstanding payments discounted at the treasury rate plus a premium. In comparison to the call option, the defeasance option does not expose the investor to reinvestment risk. Finding a new investment opportunity might not be attractive to the bondholder, especially in a low interest rate scenario where calling might be more beneficial over defeasance to the bond issuer. In contrast, a defeased bond exactly replicates the expected cash flows of the bond without risk of default.

Our findings show that when examining only bonds that carry a defeasance option, bonds with call options are not significantly less likely to exercise a covenant defeasance option. This supports the view of Bienz et al (2013) that calling of bonds does not substitute for defeasance, but does not conclusively prove that there is no correlation.

Bienz et al (2013) show that there is a positive association between the number of covenants in a bond, and the inclusion of a defeasance option. The intuition is that

companies are willing to accept more restrictive covenants if they can be removed ex post. Expanding on this intuition, we believe that among bonds with the option to defease, the number of covenants positively affects the chance of exercising defeasance options. This is reasonable as companies that are more restricted can have a higher chance of encountering situations where the covenants limit value-adding corporate actions.

In the comparison of our data findings with the FISD data, we found that the number of covenants carried by a bond is positively associated with the probability of a defeasance option being exercised. The results are significant even when adjusted for the higher number of covenants in the bonds with a defeasance option. This is in accordance with our expectations.

Restrictive covenants will potentially limit the possibilities of a company to act as they wish. Value-adding corporate actions may be restricted by the covenants of their bonds. As covenant defeasance exercise is not without cost, we believe that defeasance will often be exercised in conjuncture with major corporate action. This is because a major value-adding action is required to justify the cost of defeasance. Our findings show that defeasance exercise is often jointly observed with other major corporate actions. 65% of the defeasance exercises had associated major corporate events. The most frequent actions were mergers, acquisitions and refinancing.

Within this thesis, we document existing theory and major previous literature used in section 2 and 3. All the steps used in building our dataset of defeasance exercise, and the associated software needed is documented in section 4. In section 5, we test the predictions presented in the introduction, using regressions and descriptive statistics. Section 6 concludes our findings.

2 Institutional Background

2.1 Corporate Bonds

A corporate bond is an exchange traded fixed income security. It makes regular coupon payments and returns its face value at the final payment date.

As long as the bond-issuing company is liable to the bondholders, the bondholders are exposed to the risk that the bond issuer might not be able to pay back the agreed payable amount between the parties (Bodie, Kane, Marcus, 2011).

What firm specific risks a company carries, is largely a matter of a management's current and future strategic and financial decisions. In most lending, there is also a potential for agency problems. Agency problems can arise when there is information asymmetry and when one entity's outcome depends on a different entity's actions on behalf of the first entity. When the latter entity is maximizing its own benefit at the expense of the former, an agency problem arises (Pindyck & Rubinfeld, 2005).

Brandon (2013) states that "When a firm adds risky debt to its capital structure, it introduces a series of financial obligations, legal constraints, and incentives that can cause conflicts between managers, shareholders and debt holders." Myers (1977) showed that when a firm has risky debt in its capital structure, managers acting in the interest of shareholders might reject positive net present value investment opportunities. This underinvestment or "debt overhang" problem occurs when a positive net present value project decreases the value of equity because some of the value created goes to the debt holders.

The inclusion of covenants is a common way to remedy these problems. Covenants are usually action restricting, which limits certain actions that might increase the bondholders' risk of not being paid their full coupons and face value. Common covenants are dividend restrictions, subordination of further debt, security through collateral and change of control (Smith & Warner, 1979).

In some occurrences, companies are faced with potential value-adding actions like refinancing because of interest rate changes or expansions through positive net present value opportunities. Restrictive covenants like limitations on debt, changes in control, or similar, might hinder the company in executing these actions. These firms will therefore want to renegotiate the covenants of their debt to execute these value-adding actions. However, renegotiating covenants of publicly traded debt is very difficult and costly. Bradley and Roberts (2004) state that renegotiation of public corporate debt is virtually impossible. The Trust Indenture Act of 1939 states that a two-third approval requirement of the bonds not owned by the issuing company is necessary to remove covenants.

A way to remedy a difficult covenant renegotiation situation is to buy back all the debt owned by bondholders. If the firm manages to buy back the issue, renegotiating is no longer a problem since the company now owns their own debt and can do as they like. This may be a cheaper and less time consuming way than renegotiation. Indeed, Brandon (2013) finds in his research paper that the primary motivation for debt buy-backs are to ease debt induced investment frictions.

2.2 Debt Repurchases

There are several ways to buy back debt. Common ways are call provisions, sinking funds, convertible provisions and tender offers (Fabozzi, 2012).

A call provision is an included option, which gives the right but not the obligation to buy back bonds at a specific date at a specific price, usually set above the bonds' face value. A sinking fund is a more gradual way to repurchase bonds. The company deposit funds into a sinking funds account administered by a trustee that repurchases the bonds in the open market. Convertible provision is an option where the company can convert the bond debt into equity with a pre-specified exchange price. In addition, tender offers are often used. This is a bid to all the bondholders to sell back their bonds to the bond issuer at a price usually set above the quoted market price.

2.3 Covenant Defeasance

An alternative way to remove covenants is the inclusion and exercise of a covenant defeasance option. Covenant defeasance or "legal defeasance" is an option that is frequently added to corporate bonds (Bienz et al, 2013). As illustrated in figure 1 below, the option allows the bond issuer to replace the bond issuer's obligations to pay the coupon and principal to a pre-paid and closed off escrow account. The escrow account is administered by a bank on behalf of the depositor.

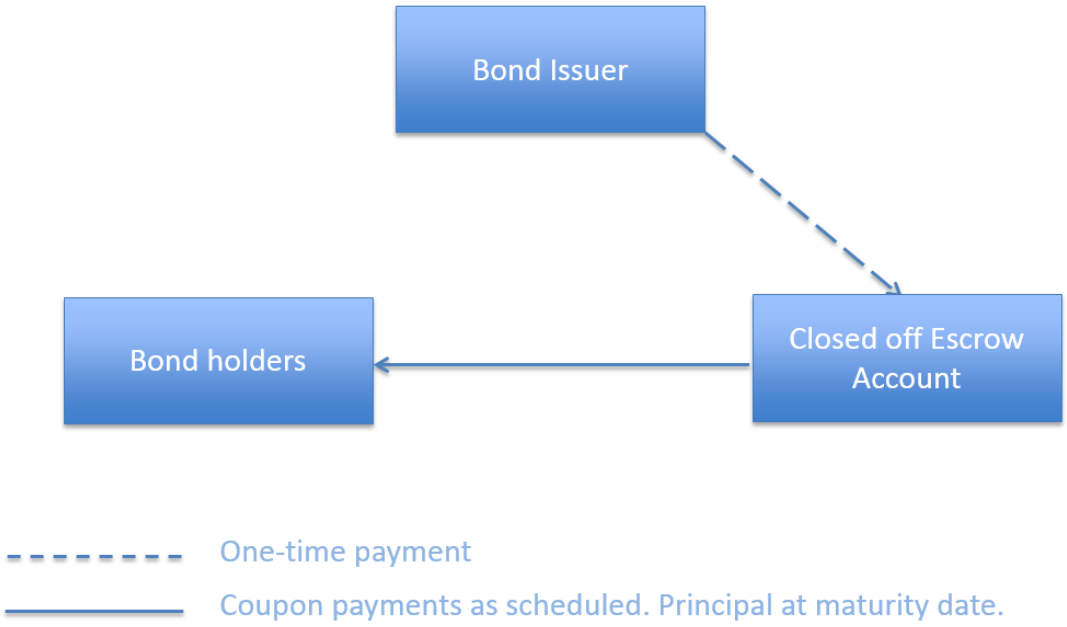

Figure 1: Illustration of the cash flows of a bond after a covenant defeasance option is exercised

As the amount is pre-paid and restricts access for the bond issuer, the bondholders will receive the expected payments from their purchased bond. In addition, there are no tax consequences for the bondholders. The reason for this is that there is no gain realized for the bondholder at the point of defeasance exercise, since the bonds are not sold. By guaranteeing their promise to pay the coupons and the principal of the bond, issuers can detach themselves from covenants that restrict management from executing plans that are in the company's best interests (Mergent, 2014).

In modeling terms, covenant defeasance will change the pricing of the bond to the following:

$$\boldsymbol{P_0} = \left(\frac{\boldsymbol{C}}{\left(\boldsymbol{1} + \boldsymbol{r_f}\right)^{\boldsymbol{1}}} + \frac{\boldsymbol{C}}{\left(\boldsymbol{1} + \boldsymbol{r_f}\right)^{\boldsymbol{2}}} + \cdots + \frac{\boldsymbol{C}}{\left(\boldsymbol{1} + \boldsymbol{r_f}\right)^{\boldsymbol{n}}} \right) + \frac{\boldsymbol{FV}}{\left(\boldsymbol{1} + \boldsymbol{r_f}\right)^{\boldsymbol{N}}}$$

Where:

P_0 = Market value of corporate bond after defeasance

C = Coupon payments

r_f = Risk free spot rate

N = Years to maturity from today

n = Specific year between present date and maturity date

FV = Face value of the bond

In comparison to a regular corporate bond, the difference is the discounting factor of the coupons and the face value. The discount rate "i" has been replaced by "r_f" which denotes the risk free rate for each period. This is done since the bond needs to be considered risk free for the covenant defeasance to be effective.

2.4 Other Terms Related to Defeasance

Terms that are frequently used along covenant defeasance (or "Legal defeasance") are in-substance defeasance and Economic defeasance.

Economic defeasance is similar to covenant defeasance as the coupons and face value for the issued bond are secured in a closed off escrow account. While it has the effect of removing the bonds from the balance sheets of the company performing the economic defeasance, it will not free the company from its covenants of the bond. This is also known as in-substance defeasance.

2.5 Potential Motivation to Defease

Restrictive covenants on bonds might restrict firms from pursuing value-adding actions. Major corporate events have the potential to change the capital structure and

key bond covenant financial measures. Removing such covenants through a covenant defeasance will enable the firm to pursue previous covenant restricted corporate actions.

Another reason to defease might be that a bondholder is speculating that the bond issuer wants remove the bond's covenants. Since such an action requires the consent of bondholders, these might be able to block such efforts by refusing to accept repurchase offers or re-negotiation of the covenants. By doing so, the bondholders can hold the bond issuer "hostage" and demand a price for their bonds that is higher than market value. This is known as a "Hold up problem". The inclusion of defeasance options can limit hold up problems (Bienz et al., 2013), but it may be necessary to exercise the covenant defeasance option to remove hold-out investors in some cases.

Kahan and Rock (2009) show how investors may aggressively pursue bonds where the covenants are breached, and sanctions have not been enforced. By enforcing breached covenant terms, they can force companies to immediately repay the bond. Exercise of covenant defeasance may be a way to remove such troublesome investors. A breach of covenants that triggers default requires a cash payout of the outstanding coupons and face value and often triggers legal fees. Since a riskless replication is sufficient for defeasance, this might suggest that defeasance is less costly. However, it is not clear if the cost of exercising a covenant defeasance is less than the cost of managing such investors.

3 Previous Literature

3.1 How to gather data using a web crawler: An application using SAS to search EDGAR

This paper by Joseph Engelberg and Srinivasan Sankaraguruswamy (2006), discusses how to use the analytics program Statistical Analysis System (SAS) to gather and search data from EDGAR (the SEC database). It also includes a complete copy of the program that Engelberg and Sankaraguruswamy have written to perform searches (henceforth called the "SAS program"). This paper provided inspiration for our search program used in this thesis. An important piece of information gathered from this paper is an alternate download link that uses the HTTP protocol. The SEC specifies a FTP download link that provides significantly lower download speeds due to the need to authorize each file for download.

There is no use of the actual code from this paper as it is written in SAS, whereas our program is written in C#. Because data is gathered from the same source, there are several similarities in how the programs work. However, there are some key differences:

1. The SAS program is more geared towards doing searches on a known subset of companies, although it can do searches on all companies. Functionality to search a known subset of companies has not been implemented, as it has not been needed for our purposes.
2. The SAS program downloads the forms that are requested for searching each time a search is made. After the search is made, the data is disposed, and will need to be re-downloaded if another search is made. This structure requires no storage space for the forms, and there is no lengthy download time before a search can be made. On the other hand, searches will be slower since the form transfer rate will be considerably lower from the remote servers than stored locally on a hard drive. This structure was probably the most reasonable for them, given that the program is geared towards searching

smaller subsets of known companies. In 2006, when their paper was published, the total number of all submitted forms was 4,249,586 compared with 14,036,271 forms in September 2014. In addition to an increased number of forms, the file size has increased significantly.

3. The SAS program requires the SAS software suite to execute searches and perform editing. Our program can run without any pre-installed software on modern Windows computers. To make changes to our program, Microsoft Visual Studio is required. Due to SAS missing important embedded methods compared to C#, and the inability to create a standalone program, it was less suitable to the needs of this project.
4. The SAS program requires the user to download, merge and manipulate the form metadata. The SRM5K has simplified this process and will automatically download, parse and save the information at the press of a button. The SAS program does offer the user the ability to manipulate the dataset before a search, provided they are familiar with the SAS programming language. This functionality is not included in our program, but can be added by a user proficient in SQL and C#.

At present, the SAS program does not work without modification due to changes in how the index files are structured by the SEC since the SAS program was written. It has been written to parse index files using fixed column width, whereas index file columns are now split using the delimiter "|".

3.2 The Defeasance of Control Rights

This paper by Carsten Bienz, Antoine Faure-Grimaud and Zsuzsanna Fluck (2013), discusses how the implementation of covenant defeasance can substitute for the renegotiation of bond terms. Their findings are as follows (direct quote from abstract):

1. With the option to remove covenants, issuers are willing to accept more action-limiting covenants ex ante.
2. The exercise price is set high enough so that the option is only exercised in the good state.

3. Financially constrained firms with high growth opportunities and higher degree of uncertainty are more likely to include this option.
4. Investors trade off the yield for reduced risk upon exercise in the good state and higher number of covenants in the bad state.
5. Investors accept a lower yield on bonds with the option to remove covenants even after controlling for the number of covenants.

The paper focuses on the effects on bonds that include a defeasance option, versus ones that do not. We wish to focus on bonds where the option is actually exercised. The paper has been a major inspiration for our thesis. The following points from this paper are incorporated into our thesis:

1. Findings indicating that call options do not substitute covenant defeasance due to Make-Whole provisions and risk of reinvestment.
2. Regression results showing that the number of restrictive covenants is statistically significant and positively linked to the probability that a bond includes a defeasance option.
3. A theory that activist investors that pursue under-enforced covenants as described by Kahan and Rock (2009) may be dissuaded by covenant defeasance.
4. The use of data from the Mergent Fixed Income Securities Database can be used to complement our gathered data on defeasance in regressions.
5. A theory that the inclusion of covenant defeasance option can limit hold-up problems where activist bondholders can resist value-adding corporate events requiring covenant removal or renegotiation to attain a higher return for themselves.

4 Data

To the best of our knowledge, there exists no comprehensive database of covenant defeasance option exercise. Mergent's Fixed Income Securities Database (FISD) lists only 11 examples. There are other examples mentioned in Bienz et al. (2013) such as Aleris, but none of these examples corresponds to the ones given by FISD. Bloomberg does not seem to distinguish between called and defeased bonds.

Hence, we needed to crawl EDGAR in order to examine corporate filings. Using our self-developed search program, we are able to analyze the contents of 1,233,691 8-K and 152,076 10-K forms for covenant defeasance exercise.

In the following section, we outline the steps used in setting up our program and using it to create the dataset.

4.1 The Search Program

The following section is a cursory introduction to the program. The code of the main program components, as well as technical details on various components can be found in the appendix. We recommend that anyone wishing to alter the code of the program should study the information in the appendix. An overview of certain IT-terms that has been used in this section is also available in the appendix.

4.1.1 Overview

SEC Resource Manager version 5K (SRM5K or "the program") is a program designed to search through the text of any form that has been submitted to the SEC database (EDGAR). The program performs all the steps needed to facilitate such a search with a minimum of user input. It has a user-friendly interface that requires no programming or database knowledge, which makes the program easy to use for a variety of users. The search results are provided as output in datasheets in the comma separated value (.csv) format, which is readable by most data manipulation software.

The program has been designed to operate from an external hard drive. The only prerequisite is .NET Framework 3.5 installed on the computer. Newer versions of Microsoft Windows will usually have this pre-installed, and will install it automatically if this is not the case. Users wishing to make changes to the program code need to have Microsoft Visual Studio 2008 or newer installed.

The program has an offline structure that requires large amounts of storage space. If the program is copied, the different copies of the program are not necessarily consistent. The program independently assigns a primary key to each record. If not every instance of the program parses the exact same index files in the exact same order, differences can arise. This means that the downloaded forms from one hard drive cannot be used with the result file from another. All forms are still downloaded, and users can alternatively use SEC accession numbers as a primary key.

4.1.2 Disclaimer and Distribution

Users are permitted make changes to the program as long as the original authors are sufficiently credited. The names of the original authors should always be visible on the startup screen of the program. Additional authors can claim credit as long as it made clear to the user which changes they made. The authors must authorize any commercial use of the program or the information it generates. Any commercial use must adhere to the terms of use of all constituent content of the program.

Should anyone wish to duplicate the program, one can simply copy the entire contents of the hard drive containing the program to a new drive. One might want to format the contents of the new drive before copying to avoid any producer-installed software from interfering with the program.

4.1.3 Hardware Requirements

The program is designed to run from an external hard drive. This is done because the forms in aggregate will use a significant amount of storage space. As more forms are added to EDGAR with time, the amount of storage space needed will increase. At the

time of writing, the storage requirements are about 400GB per major form type (such as 8-K and 10-K forms). We expect storage requirements to increase by about 50-150 GB per additional year of forms downloaded of each form type. For other less frequently used form types, the storage requirements are significantly smaller. The only formal requirements are that the database file and the folders containing the forms must be in the <root>:\EDGAR folder of the hard drive the program is stored.

There are no minimum requirements for the computer running the search. Any reasonably modern Windows computer should work. Less than 8 GB of installed and usable RAM might create problems in the future, due to the increasing size of individual files submitted to the SEC. 8 GB of RAM should therefore also be considered a minimum, especially when working with large forms such as 10-Ks.

The main concern for the search speed of the program is the read speed of the hard drive being used. The computer and the external hard drive should therefore be USB 3.0 compatible or better as this greatly enhances search speed. Solid-state drives should offer a major performance benefit over traditional hard drives, and should be considered for users in need of increased search speed.

4.1.4 SRM5K Program Components

The program can be divided into 5 distinct processes as shown in figure 2 below.

Figure 2: A simplified process description of the program

We found this method of dividing the necessary procedures of the program to be the most logical. Hence, it is therefore also how the code is structured into separate units. The description of each code block (method) is based on this structure. The entire

program solution contains several additional modules, which are not described, that does supporting operations and maintains the user interface. These are described in "Additional Helper Procedures" in the appendix, and the supplied program source code.

4.1.4.1 Method for Downloading Form Metadata from the SEC Website

The downloading of index files from the SEC website requires four distinct steps as outlined in figure 3.

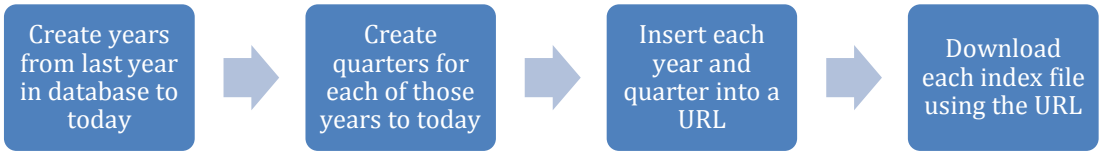

Figure 3: A simplification of the process necessary to download index files.

When downloading forms, it is necessary to know their address on the SEC website. Fortunately, the SEC supplies quarterly files containing information on all the forms made available on their webpages. The information available in these files is:

- Company Name
- Form Type
- Form Submission Date
- CIK-number
- Link/server location

This information is distributed by the SEC in files named "company.idx" on their FTP server. The .idx format is a text format data files. They can be opened using any text viewer, and the information is stored in plain text. The index file is stored using the following format:

ftp://ftp.sec.gov/edgar/full-index/2002/QTR1/company.idx

This is the location of the index file for the first quarter of 2002. Because the format of the link stays the same for all years and quarters, we can design code that alters the URL for downloading each index file. The code will increase a number starting at 1993 and the term to be met is that the number is equal to this year. This creates a list of

numbers representing each year between 1993 and the current year. For each year, quarters are created and labeled from 1 to 4. For the current year, quarters are only created until the current quarter. This is inserted into the URL template, and used to download each SEC index file.

The file is renamed at downloading to the format <Year>-<Quarter>.idx. The file for the first quarter of 2002 will therefore be named 2002-1.idx. They are downloaded to the folder "MASTERINDEX" on the hard drive containing the program. Note that every time the procedure for downloading index files is run, the contents of the "MASTERINDEX" folder will be deleted before downloading new files.

4.1.4.2 Method for Parsing Form Metadata into Memory

The steps outlined in figure 4 are required to read the index files. Note that this procedure does not complete a process on its own, and it must be combined with the saver in section 4.1.4.3.

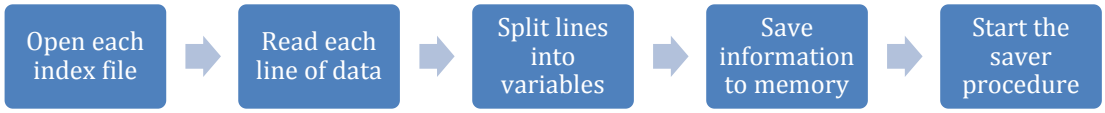

Figure 4: A simplified process chart for parsing form metadata.

In order for the program to use the information contained in the downloaded index data files, the information needs to be parsed into a machine-readable format. The downloaded files are in the .idx format, which is readable in visual studio using the embedded "streamreader"-function. The program opens each file in the folder of index files. It reads the file line by line until it encounters a line of dots. This is a data anchor designating that the header of the file has ended and that subsequent lines contain data. The program will then go through each item until it reaches the end of the file.

The data item can be in different formats depending on when the files where released from the SEC. The program supports index file formats back to at least 2006. All files downloaded from the SEC will be in the newest format.
The current data format uses a symbol delimited format where "|" is the delimiter. The data is stored in the following order:
CIK|Company Name|Form Type|Date Filed|Filelink
A typical data line will therefore look like this:
1000180|BOEING|8-K|2014-01-22|edgar/data/1000180/0001000180-14-000007.txt
The program will split each line on the delimiter and store each item in a pre-defined object class called DocumentInstance.

The object class contains a variable called IndexID that is not supplied with the index file from the SEC. This is a number that is generated by the program to give each form information item in the database a unique identifier (primary key). This is also the key used to name the forms when they are downloaded.

By forcing information to adhere to a set specification in the initial parsing process, miss-parsed information can be identified before reaching the database insertion phase. This adds robustness by reducing the danger of adding erroneous data to the database, especially since our selected database engine does not have a dedicated date format.

4.1.4.3 Saving Form Index Data

The steps outlined in figure 5 will save the index data that is parsed into memory in part 4.1.4.2.

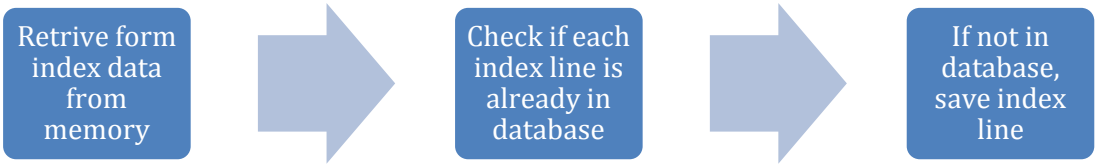

Figure 5: The process of saving the index data to the database

Once the entire file has been read through, the database saver method is invoked. It will go through each parsed item in the local memory, and check if it is already present in the database. The item FileLink is used as a candidate key to determine if the record is already present, since no other combinations of the data are robust enough to be a candidate key. This is because the same company can submit two forms of the same type in a single day.

The matching procedure is very time-consuming. This is partly because the database engine lacks string-indexing capabilities, and partly because the matching procedure prioritizes robustness and simplicity over speed. For example, to control the integrity of the entire database, the program would have to make about 196 trillion matching operations (14 million existing items multiplied by 14 million potentially unknown items that need to be controlled). For each record found by the parser, the program makes an SQL-query asking for a record with the same FileLink as the record to be inserted. If a match is made, nothing is inserted, as the record already exists.

If no match is found the program prepares to insert the information. The information is parameterized, which is a method of converting data items in a program to SQL-database format before the transaction with the database takes place. This is generally considered best practice as it reduces vulnerability to SQL-injection attacks, and makes the SQL interchangeable between different database systems (Feuerstein, 2007). This could be useful if one would like to upgrade to a different database engine that gives higher search performance.

A method of database insertion is used where changes are not finally saved until the code has sent a signal to the database that all rows have been successfully inserted. This means that if an error occurs while saving the data, all insertions made are rolled back, and the database remains unchanged. This reduces the risk of records being improperly inserted, and therefore increases robustness.

4.1.4.4 Method for Downloading Forms to Local Storage

Figure 6: A simplified model of the steps needed to download forms from the SEC database

This procedure uses the saved form index data to download the actual form to local storage using the steps outlined in figure 6. It will download all forms of a selected type between 1993 and the newest date in the index database. The files are downloaded to the following location:
<Drive letter of drive the program is launched from>:\EDGAR\<Form Type>\<Year form was submitted>\<IndexID of form>.txt
Therefore, a 10-K form with submission date 23.08.2008 and assigned index id 3856300 will be downloaded to C:\EDGAR\10-K\2008\3856300.txt when the program is stored on the C: disk.

Downloading forms will be time consuming. Larger files (like 10-Ks) are faster to download per gigabyte than smaller files (like 8-Ks). This is due to the slight time the SEC database needs to handle each request. When downloading 10-Ks, the authors have been able to download at close to the max speed of our available network (about 1.6 Mbits/s). Still, due to the amount of data, users should expect downloading a single recent year of one form type to take several days.

In the program, all forms are download to local permanent storage, before any search can be made. This opposes the solution chosen in the SEC-scraper made by Engelberg & Sankaraguruswamy (2006), who download the relevant forms each time a search is made, and disposes the data after the search is complete.

The reason the data is stored locally is that the amount of data has increased markedly since the Engelberg and Sankaraguruswamy wrote their SEC-scraper in 2006. As

shown in figure 7, this is especially true for 10-K forms. All 10-K forms from 1993 to 2006 sum to 55.88 GB while the 10-K forms for 2013 alone sum to 95.8 GB.

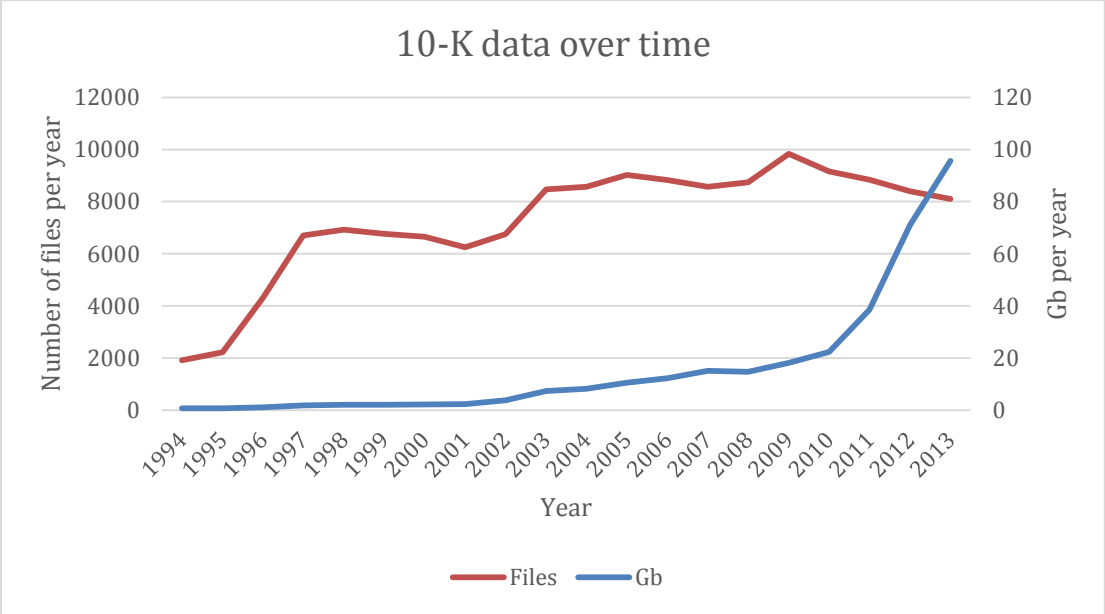

Figure 7: Comparison between the number of forms and the total size of all forms per year for 10-K. Source: SRM5K

The reason for this increased amount of data is partly increased file sizes. A change can be found around 2002-2003 when submission of HTML-forms rather than text forms became more common. A marked increase was around 2010, as a consequence of multimedia content being attached to forms, as shown in figure 7. This multimedia content can be pictures, PowerPoint presentations, video etc. The program has no method of searching through this content, as each format would require decoding from raw code and then a codec to interpret the data. While the multimedia content does not improve searches, and consumes a significant amount of storage space, it was decided not to make any effort to remove this content from the form files. We decided to keep the downloaded data identical to the data on the SEC servers. 8-K forms have also increased in size from around 2010, as seen in figure 8. This increase is less pronounced than for 10-K forms.

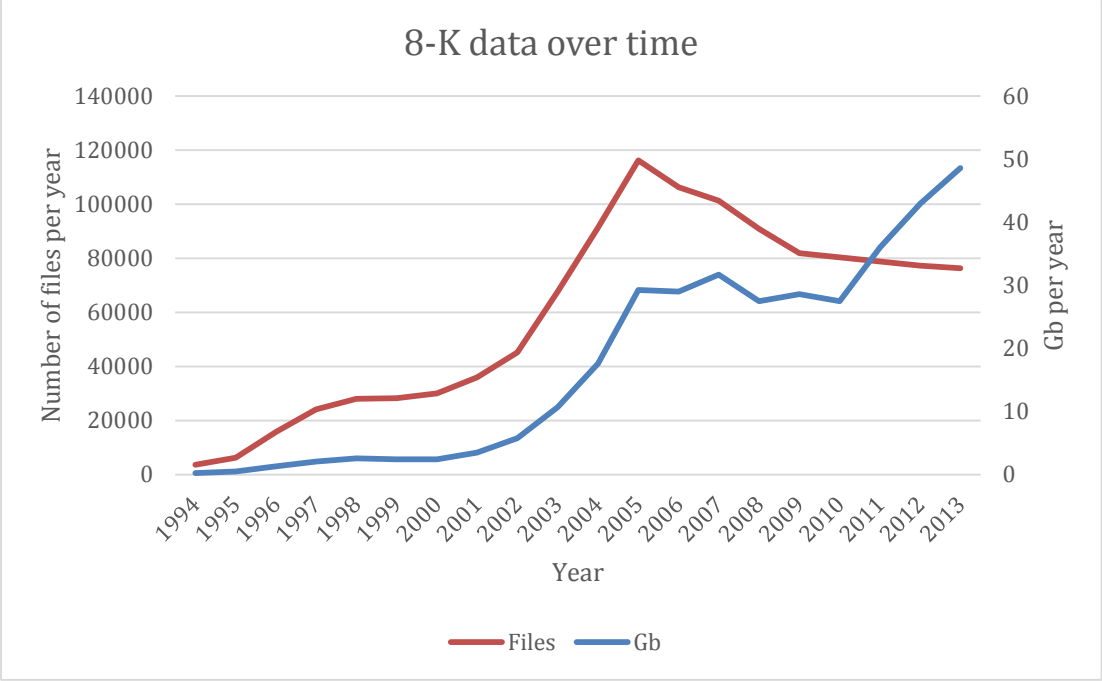

Figure 8: Comparison between the number of forms and the total size of all forms per year for 8-K. Source: SRM5K

Users should note that the program will only download forms between 02.00 and 11.00 UTC. This is due to a request from the SEC that bulk downloads should be done outside working hours, defined as between 9PM and 6AM Eastern Standard Time. The SEC does not factor in daylight saving times, so neither does the program. If a download is initiated within US working hours, the program will pause downloading forms, and display a message explaining why. Downloading will start automatically at 02.00 UTC, and pause again at 11.00 UTC. An override of the restriction is implemented for users who wish to download only a few files.

The program will indicate to Windows that it is currently performing operations, and that it should not enter sleep mode. In practice, this method has proved somewhat unreliable, as the computer will frequently initiate sleep mode anyway. An alternative method of preventing sleep mode is to keep a window of a media player such as VLC open while performing operations or the user can manually deactivate sleep and hibernation modes in Windows.

4.1.4.5 Method for Searching Through Downloaded Forms

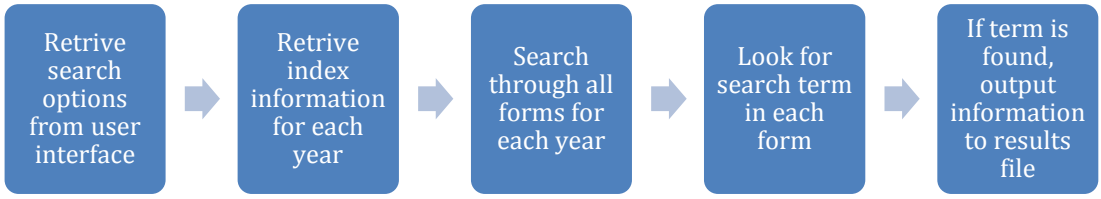

Figure 9: The simplified steps used by the program to search through SEC forms

Using the steps from figure 9, this method will go through each of the forms selected for search through specification in the user interface. It returns a list containing the hits made.

The user can define their search in the user interface. In the input line, the user may input one or more distinct search strings. The user should note that the program searches the form for the set of input characters in their exact order (string). This is opposed to search engines such as Google, which identify whole words. The reason for searching for strings rather than words is the large amount of extra code needed to differentiate words from whitespaces and HTML-code. What the user reads as a space or newline will be one of a number of different encoding options. It would also require a robust HTM-decoder, to avoid mistaking search text for code. One possible effect of this is that the program will return hits for search strings that are part of another word. For example, a search for “Invest” will yield a hit when encountering the word “Investment”.

The search procedure is not case sensitive. This is currently hard-coded into the program, and can be changed by either recoding the program or altering the program to make case sensitivity an option in the user interface.

The user must select the form they wish to search. A search may only be made on one form type at a time. If one wishes to make searches on multiple form types, one must perform multiple searches and merge the results. Although it is technically possible to

search multiple form types in one search, it has not been a prioritized feature, since it would require a substantial amount of additional code.

It should be noted that the user can select forms for search that are not present on the hard drive. The user should therefore download the desired form type in the update tab before searching to insure its presence. A search made without the forms present will end prematurely without returning any hits.

The search is made chronologically. The program will split the forms to be searched by year, and only searches one year at a time.

The results of the search are returned in a file named “results.csv” that is stored in “<root>\EDGAR\results.csv”. A copy of files where a hit for the specified search term was made is saved to a folder named “RESULTFILES” that can also be found in the “EDGAR” folder. Note that both the result file and folder is cleared each time a search is initiated, so users should save their results elsewhere after a search has been made.

4.2 Scope of the Data

This thesis is limited to US corporate bonds since it is a large homogenous market. The EU is also a large market, but US financial legislation is more similar across regions than in the EU, and US bonds will therefore be more comparable. US corporate bonds will also have a single currency, which adds to comparability. When using the search program we chose to focus on 8-K and 10-K filings. We used these, as all significant transactions in a company that affects stakeholders are required to be disclosed in these filings.

4.3 Sources

The sources for our data are primarily the Securities and Exchange Commission’s (SEC) Electronic Data Gathering, Analysis, and Retrieval (EDGAR) system. This system stores all submissions by companies and others who are required by law to file forms with the SEC. The SEC states that the primary purpose of the database is to increase the

efficiency and fairness of the securities market for the benefit of investors, corporations, and the economy. This is done by accelerating the receipt, acceptance, dissemination, and analysis of time-sensitive corporate information filed with the agency. It is important to note that the EDGAR database's filings only date back to 1993 or 1994 in some instances (SEC.gov, 2014). This database has been the underlying data for our searching using SRM5K.

In addition to EDGAR, the Bloomberg financial database was used to triangulate results and add data to the findings. Bloomberg L.P. is the company that owns and services the Bloomberg financial database. The database is extensive and provides both broad and in-depth data about most types of assets classes including equities, government and corporate debt, money market securities and commodities. In addition to general information about the different securities collected from SEC-filings, the database also provides information based on external sources like major and reputable newspapers (Bloomberg.com, 2014). The reason for our addition of this database is its structured qualities and ease of use regarding look-up of specific securities.

The Fixed Income Securities Database (FISD) is a database owned and maintained by Mergent, which is a leading provider of business and financial information. FISD contains issue details on over 140,000 corporate, corporate MTN (medium term note), supranational, U.S. Agency, and U.S. Treasury debt securities and includes more than 550 data items. FISD provides details on debt issues and the issuers, as well as transactions by insurance companies. It is used to examine market trends, deal structures, issuer capital structures and other areas of fixed income debt research (Mergent.com, 2014).

In addition to EDGAR, Bloomberg and FISD, news services like businesswire.com and prnewswire.com was used to identify significant corporate events.

4.4 Search Iteration and the Data Gathering Process

In this section, the work method to identify covenant defeasance option exercises and compile additional data about these findings is explained. An overview of the steps in the work process is illustrated in figure 10.

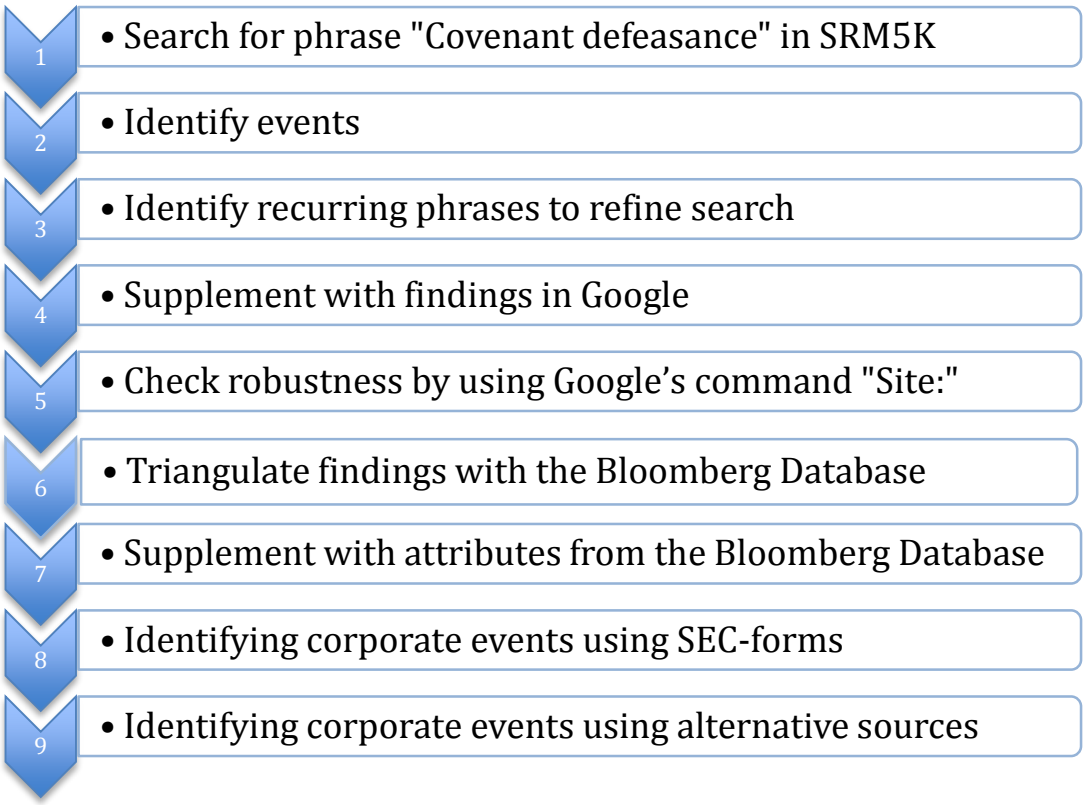

Figure 10: Work processes when compiling the covenant defeasance exercise dataset

In the initial open search for "covenant defeasance", we expected that some of the returned results would not be valid occurrences of covenant defeasance. We defined a "false positive" as findings that were not a valid covenant defeasance exercise and "true positive" as a search result that was a valid covenant defeasance exercise.

Initially, the search word used in the SRM5K to find events of covenant defeasance was "Covenant defeasance". By doing this, 5939 hits of the 1.4 million forms were found. When looking through these results, it was clear that most of these findings were bonds that included a covenant defeasance option and were not an option exercise. These false positives made it hard to identify the true positives. However, by manually searching through some of the hits, some true positives were identified.

In addition to returning forms with the search word, the SRM5K also returns the heading of its hits. We tried to identify headings that could indicate an exercised defeasance option, but were not able to find any that consistently was used for describing covenant defeasance exercises.

Since the wordings in the SEC-filings are often standardized, we checked some of our confirmed true positives to identify standardized phrases. One recurring phrase found in three of the true positives, was “Effected a covenant defeasance”. Other less frequent phrases was “Executed a covenant defeasance” and “Exercised a covenant defeasance”. When focusing the search using these phrases, the hits generated in the SRM5K where mostly true positives.

In addition to the searches in SRM5K, Google searches where used. The main strategy was to start using the focused search phrases “Effected a covenant defeasance”, “Executed a covenant defeasance” and “Exercised a covenant defeasance” found earlier. When doing these searches, a number of false positives were returned. To resolve this, commands in Google for exclusions of standard phrases in the false positives where used. Examples of these are “upon election”, “at any time” and “If we”. These phrases are common in texts that state the existence of a covenant defeasance option, but not an actual exercise of the option. By doing this, additional exercises of covenant defeasance were found.

Google was also used to search the EDGAR database. By using the command “site:sec.gov” in front of the search phrases, a filter is added to the search excluding all hits not located at the site “sec.gov”, which is where the EDGAR-database is located. When using this command on previous identified covenant defeasance phrases, the additional findings were limited. However, previous hits from the focused SRM5K-search were found. These findings were mixed with noise from numerous false positives. The fact that no new hits where made, indicates that the search procedures used in the SRM5K search where robust.

After having found covenant defeasance events, the Bloomberg database was used to validate the results. If a unique identification number for the bond was not included in the form or statements from the company, the findings were cross-checked with other information about the bond to identify the correct bond name in the Bloomberg database.

Since the SEC-forms where the covenant defeasance exercise was stated, did not contain complete data about the bonds, data from the Bloomberg database were used to supplement our dataset. From this database, data such as Employer Identification Number (EIN), Committee on Uniform Security Identification Procedures (CUSIP), bond class, face value, industry classification, issue date, maturity and coupon rate was retrieved. Under the category "corporate events", information about tender amount, tender announcement date, tender effective dates, information about buy-backs and other information relating to the tender was found.

The Bloomberg database does not explicitly label defeased bonds as defeased. In most cases, the bonds are specified as "called" on the defeasance date noted in the corporate filing. This specification was consistent with the defeasance date of our findings. Since the defeasance dates in the corporate filings and the Bloomberg call date match, there is reason to believe that the "call"-classification is the defeasance date.

Some of the defeased bonds that are listed as called are also defined as "defeased" in a text field called "Bond description notes". This was considered as a potential source of uncovering additional defeasance hits. After consulting with Bloomberg terminal support, we were informed that doing a specified search isolated in the "bond description notes" was not a feature supported by Bloomberg at this time.

Finally, data on major prior and parallel events with the covenant defeasance was collected. The primary source of information was the forms where the defeasance

exercise was found. In addition, Google searches were also used for finding events for each company. These searches were limited to the months around the covenant defeasance date.

It is important to point out that without our self-developed search program, the true positive findings would be far less extensive. The data obtained from Bloomberg could only be extracted from the Bloomberg database after being pinpointed by SRM5K. The Bloomberg database is extensive, but is constrained by a user interface that does not allow quarries identifying covenant defeasance exercise. Solely relying on Bloomberg searches would therefore not have been feasible to create a usable dataset for our thesis.

4.5 Entity Attributions

A number of attributions for our confirmed defeasance findings were collected. These are listed, explained and documented below.

CompanyName: Notes the bond issuers company name.
CUSIP: A unique 9-character alphanumeric code that identifies a North American financial security for the purposes of facilitating clearing and settlement of trades.
EmployeeIdentificationNumber (EIN): Also known as Federal Employer Identification Number or FEIN. This number is unique for every incorporated company.
CompanyBusinessType: Bloomberg's standard industry classification.
FormType: In which form type the entity was found
FormDate: The date of which the form has been recorded in EDGAR.
CIK-Number: Central Index Key. This number is unique number the U.S. Securities and Exchange Commission gives to each company that files forms electronically.
BondMaturityDate: Date of maturity for the bond.
BondCoupon: Coupon payments in percent of face value.
BondFaceValue: The total face value of the bond.
TenderType: If the bondholders have received an offer to sell back their bonds to the bond issuer and what type of offer they have been given.

AmountTendered: The dollar-amount of the bond that the company managed to buy back of the bonds face value.

AmountTendered (%): The percentage amount of the bond that the company managed to buy back in relation to the initial face value.

BondClass: Information about the debt priority of the bond.

TenderAnouncementDate: The date a tender offer for a specific bond is announced.

TenderEffectiveDate: The date a defeasance option for a specific bond is exercised.

BondInfoLink: Notes a link to alternative attribution source.

BondInfoLink2: Notes an additional link to alternative attribution source, if applicable.

SearchWordSECResourceManager: Notes the search word used to find the entity in the SEC Resource Manager.

CorporateEvent: States if a description of a significant corporate event in the recent months around the covenant defeasance date is found. This might be acquisitions, mergers or refinancing.

TenderLink: Source of tender offer information.

CorporateEventDate: States the exact date of the corporate event.

CorporateEventLink: States the source of the corporate event finding.

CorporateEventDescription: Describes in short, the corporate event.

5 Empirical Analysis

In this section, we present an empirical analysis on the bond issuer's decision to exercise their covenant defeasance options.

5.1 Dataset and Variables

We wish to compare bonds that have and have not been defeased, to see if there are any significant variables that affect the exercise of covenant defeasance options. This is done by merging the bonds found to be defeased, with the Fixed Income Securities Database containing US-issued corporate bonds. A series of regressions are undertaken to examine if a set of variables affect the likelihood of a defeasance option being exercised. The examined variables are chosen based on potential effects found while creating the dataset, and significant findings by Bienz et al (2013) on the inclusion of covenant defeasance options.

5.1.1 Dependent Variable

The dependent variable is a binominal variable designating if a bond has exercised a covenant defeasance option and is called *Is Defeased*. The bonds that either the FISD or we have flagged as defeased have the variable set to true.

Only 21 out of the 40 bonds that were found to be defeased are present in the FISD database. Therefore, only these 21 bonds represent the positive case of covenant defeasance exercise.

Summary statistics of all the variables can be found in Table 1 in the Appendix.

5.1.2 Probit Model

The Probit regression model is used to investigate if there exists a significant relationship between an associated variable and exercise of defeasance options for predictions where regression analysis is practical.

The dependent variable is binominal, designating if a bond has been defeased. Using a standard linear OLS estimator on a binominal dependent variable is possible, but implies that the change in predicted probability is the same for all given values of X. A Probit model, which is a nonlinear probability model, is therefore used. The model measures the probability that Y=1 using the cumulative standard normal distribution function $\Phi(z)$ The Probit regression model is defined as:

$$Pr(Y = 1|X) = \Phi(\beta 0 + \beta 1X)$$

Φ is the cumulative normal distribution function and z = β0 + β1X is the "z-value" or "z-index" of the Probit model (Bienz, 2014).

The regression output is displayed in table 3 in the appendix.

5.2 Prediction 1: Defeasance options are seldom exercised

Due to the lack of trustworthy information on defeasance exercise, and the lack of reporting on the subject by major financial institutions such as Bloomberg, we hypothesize that the exercise of defeasance options is rare.

Our findings total 40 confirmed cases of exercised covenant defeasance options. Our findings range from bonds being defeased between late 1996 and late 2013. The bonds face values vary between $ 31.1 million and $ 1.25 billion with a mean of $ 278 million and a median of $ 204 million.

Of our dataset of 40 defeased bonds, we were able to join 21 of these findings with the FISD dataset. The total number of defeasible bonds in FISD is 7190, which make the defeased amount of FISD bonds equal to 0.29%. Comparing all 40 defeased bonds to the 7190 in FISD will still yield a percentage of only 0.56% defeasance options exercised. In comparison, the FISD database indicates that repurchase attempts are made on 12.07% of US bonds issued, which makes defeasance exercise seem quite uncommon in comparison.

The limited use of covenant defeasance options might explain the lack of explicit reporting of this from Bloomberg and other institutions. It might also indicate that other methods of covenant removal are attempted before a covenant defeasance is exercised. An example of this is Las Vegas Sands, which in 2002 refinanced its debt. This involved issuing new bonds and retiring existing bonds. In this transaction, the existing debt was first tendered and one of the issues failed to gain a complete tender. The defeasance option was exercised after this tender attempt failed (Las Vegas Sands, 2002).

5.3 Prediction 2: Bond issuers attempt repurchase prior to exercising a covenant defeasance option

Bienz et al (2013) suggest that bond issuers will prefer to attempt to neutralize the covenants through repurchase. However, they may need to remove any hold-out or activist investors as the bondholder can refuse any repurchase offer, potentially making a complete repurchase prohibitively expensive. Covenant defeasance may therefore be used to remove the hold-outs. We therefore expect tender or exchange offers to have a positive effect on the chance of exercising a covenant defeasance option.

By examining the forms and statements where we found covenant defeasance exercises, we found that many of the firms that exercised the option also made a tender offer prior to the exercise of the option.

Tendering	N	%
Tendered	29	72,5 %
Not tendered	11	27,5 %
Total	40	100 %

Table 1: Exhibits how many of the total bonds that were and were not tendered

Indeed, table 1 shows that 72.5% of the total issues did a tender offer of their bonds before undertaking a covenant defeasance.

Tendering Descriptive Statistics	N	Mean	Median	Highest	Lowest
Tendering %	29	74,5 %	81,2 %	99,98 %	16,6 %

Table 2: Descriptive statistics of tender offers on defeased bonds

Table 2 shows descriptive statistics of tender acceptance percentages of the tendered bonds. Bond issues usually got a high acceptance ratio, with a mean of 74.5% and a median of 81.2%. The highest acceptance ratio was 99.98% and the lowest was 16.6%. The mean was lower than the median, since most of the observations of the sample have a high acceptance ratio and some observations have a relatively low acceptance ratio.

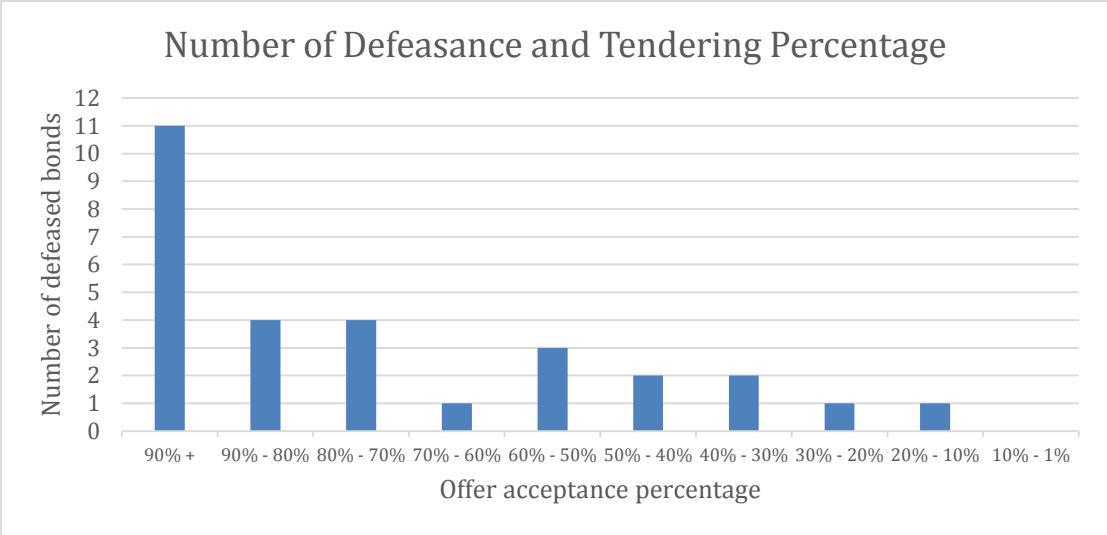

Figure 11: Illustration of tender acceptance and our defeased bond sample

Figure 11 categorizes the number of defeased bonds according to tender percentages with an interval of 10%-points for each category. Acceptance percentages above 90% dominate our sample and few bonds have tender acceptance percentage of less than 50%.

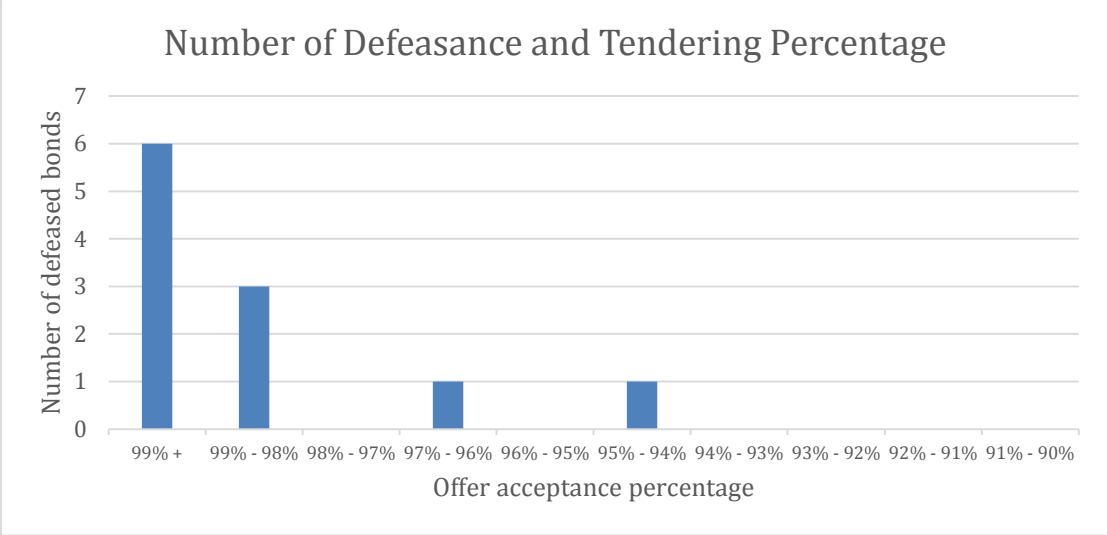

Figure 12: Number of defeasances with offer acceptance percentages above 90%

Figure 12 takes a closer look at the bonds that have a tender acceptance above 90 %, by categorizing the defeasance hits at a 1%-point interval. The majority of these covenant defeasance exercises have tender rates above 99%. Only two of the eleven observations have tender acceptance percentages less than 98%. Since all of these bonds manage to tender a large part of the issues, the cost of defeasance relative to the size of the bond issue is marginal.

Following in table 3 is the regression output of the variable *Tender or Exchange Offer,* which is a dummy variable, designating if a bond has made at least one tender or exchange offer. This variable is used to represent repurchase attempts. In specifications (1) and (4), the variable *All covenants* is omitted to test the sensitivity of the *Tender or Exchange Offer* to omitted variable bias. Specifications (3) and (6) are made using heteroskedasticity-robust standard errors. In specifications (4), (5) and (6), the datasets are restricted to only bonds that contain a defeasance option.

Specification Number	(1)	(2)	(3)	(4)	(5)	(6)
Data set	All bonds			Only bonds with defeasance options		
Tender or Exchange Offer	0.421***	0.305*	0.305*	0.341**	0.307*	0.307*
	(0.159)	(0.170)	(0.177)	(0.171)	(0.173)	(0.179)
Covenant count	N	Y	Y	N	Y	Y
Robust standard errors	N	N	Y	N	N	Y
Only defeasible bonds	N	N	N	Y	Y	Y

Table 3: The regression output on the variable indicating repurchase offers using Is Defeased as dependent variable

Table 3 shows that across all specifications, the regression coefficient is positive, which is consistent with our prediction that repurchase is positively associated with covenant defeasance exercise.

The regression coefficients are significant on at least the 90% level in every specification of the model in table 3. Specifications (1) and (4) show a higher significance level, which likely is due to omitted variable bias, because the variable *Tender and Exchange Offer* is omitted. This may indicate that the results are sensitive to omitted variable bias, and missing causal factors can therefore inflate the significance of the included factors.

Specifications (2), (3), (5) and (6) give very similar regression coefficients and standard errors. This indicates that repurchase attempts and the inclusion of defeasance options have a low degree of correlation, which is consistent with the results of the correlation matrix in table 2 in the appendix.

Although the significance of the coefficient is only 90% in some specifications, we know that the variable *Tender or Exchange Offer* used in the regression is underreported. Of the 21 instances of covenant defeasance that could be joined to the FISD database, 7 are reported as having a repurchase offer made. Our research indicates that the true number of repurchase attempts is 13 out of the 21. This

discrepancy has not been corrected in the regression dataset as it could create a bias towards the corrected data. The regression output might therefore not give an entirely accurate description of the importance of repurchase in covenant defeasance exercise.

Some bond issuers go as far as explicitly stating that the covenant defeasance option is exercised to remove remaining bondholders after repurchase. Indeed, Hudbay Minerals state in their annual information form, 27.03.2007, that they first repurchased bonds through the open market in the years 2005 and 2006. At the end of 2006, they made a tender offer for the total amount. When this tender amount failed to acquire the whole bond issue, they exercised a covenant defeasance option for the remaining amount outstanding (Hudbay Minerals, 2007). Other examples are Hovnanian Industries and Revlon Industries. Revlon first attempted an exchange offer, followed by a tender offer before the company defeased the remaining issue (Revlon, 2005). Hovnanian did an exchange offer, market buybacks and a secondary exchange offer before defeasing the remaining bonds (Bloomberg, 2014).

The conclusion is that repurchases and covenant defeasance exercise has a positive relationship. Companies repurchase bonds and finally exercise the defeasance option to remove the bondholders who do not accept their repurchase offers. These repurchase offers often have a high degree of acceptance.

5.4 Prediction 3: Defeasance exercise is more common in industries with high uncertainty regarding future financial performance

There is a possibility that companies in certain industries have stronger incentives to exercise a covenant defeasance option. Bienz et al (2013) showed that financially constrained firms with high growth opportunities and higher degree of uncertainty are more likely to include the defeasance option. This might cluster covenant defeasance exercise in industries exposed to these traits.

Industry	N	%
Aerospace & Defence	1	2,5 %
Apparel & Textile Products	1	2,5 %
Auto Parts Manufacturing	1	2,5 %
Casinos & Gaming	7	17,5 %
Consumer Products	8	20,0 %
Containers & Packaging	1	2,5 %
Electrical Equipment Manufacturing	1	2,5 %
Exploration & Production	1	2,5 %
Food & Bevarages	1	2,5 %
Homebuilders	1	2,5 %
Metals & Mining	3	7,5 %
Motion Picture Equipment	1	2,5 %
Petrolium Refining	1	2,5 %
Power Generation	1	2,5 %
Publishing & Broadcasting	1	2,5 %
Refining & Marketing	1	2,5 %
Restaurants	2	5,0 %
Retail - Consumer Discretionary	2	5,0 %
Tobacco	1	2,5 %
Transportation & Logistics	1	2,5 %
Utilities	1	2,5 %
Wireless Telecommunications Services	2	5,0 %
Sum	40	100 %

Table 4: Covenant defeasance exercise by industry classification of the bond issuer

Table 4 uses Bloomberg's industry definitions and shows that defeasance has occurred over a broad range of industries. However, two categories stand out, which are casinos & gaming and consumer products.

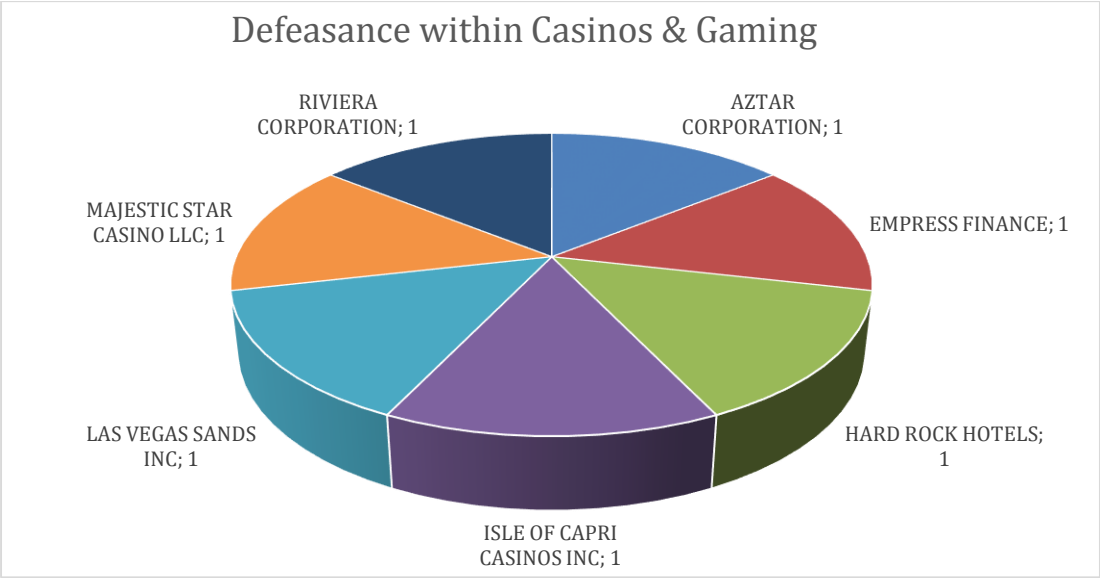

Figure 13: An overview of firms with exercised covenant defeasance options in the casinos and gaming industry

Figure 13, shows that the category casinos & gaming is populated by a diverse group of companies, with no reoccurrence among the firms. The only connection or similarity that was found was the fact that the companies are part of the same industry. It can therefore be assumed that these observations are independent.

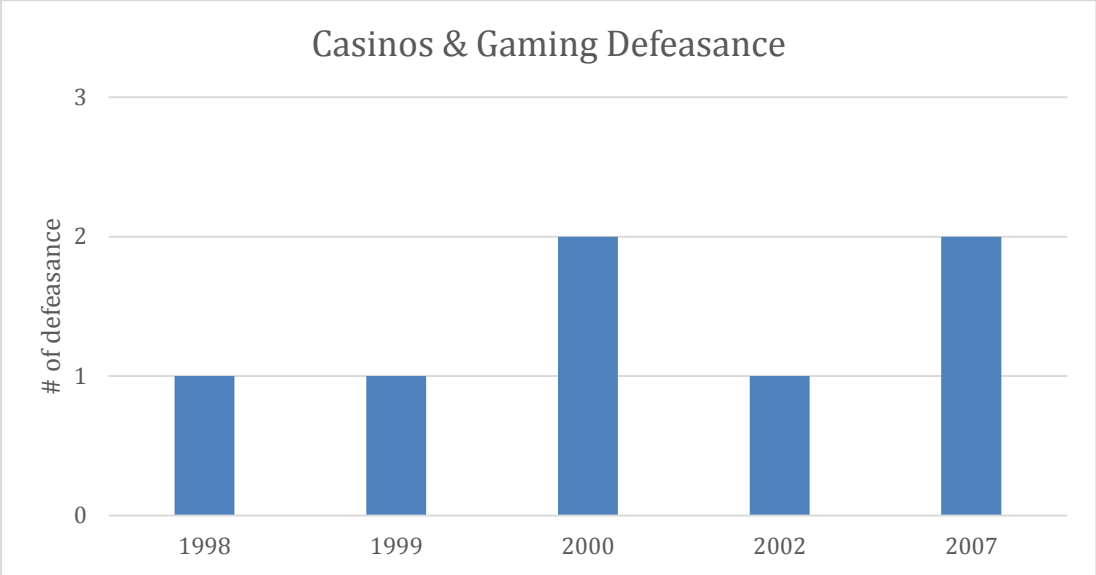

Figure 14: An illustration of defeasance in the casinos & gaming industry by year

The exercises of the covenant defeasance options in the industry over time were graphed to look for correlation with economic trends or other non-firm specific factors. As illustrated in figure 14, the observations are spread across a 9-year period and show no signs of patterns.

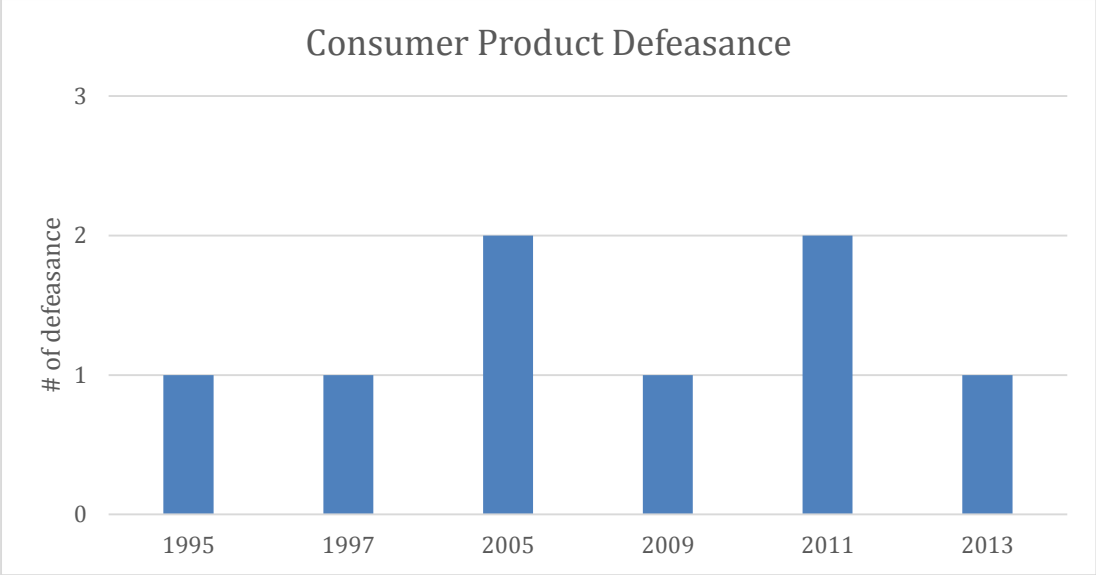

Figure 15: Illustrates defeasance in the consumer products industry

Figure 15 shows that the exercises of the covenant defeasance in the category consumer products were also evenly spread across the time period and show no indication of significant clustering of defeasance option exercises.

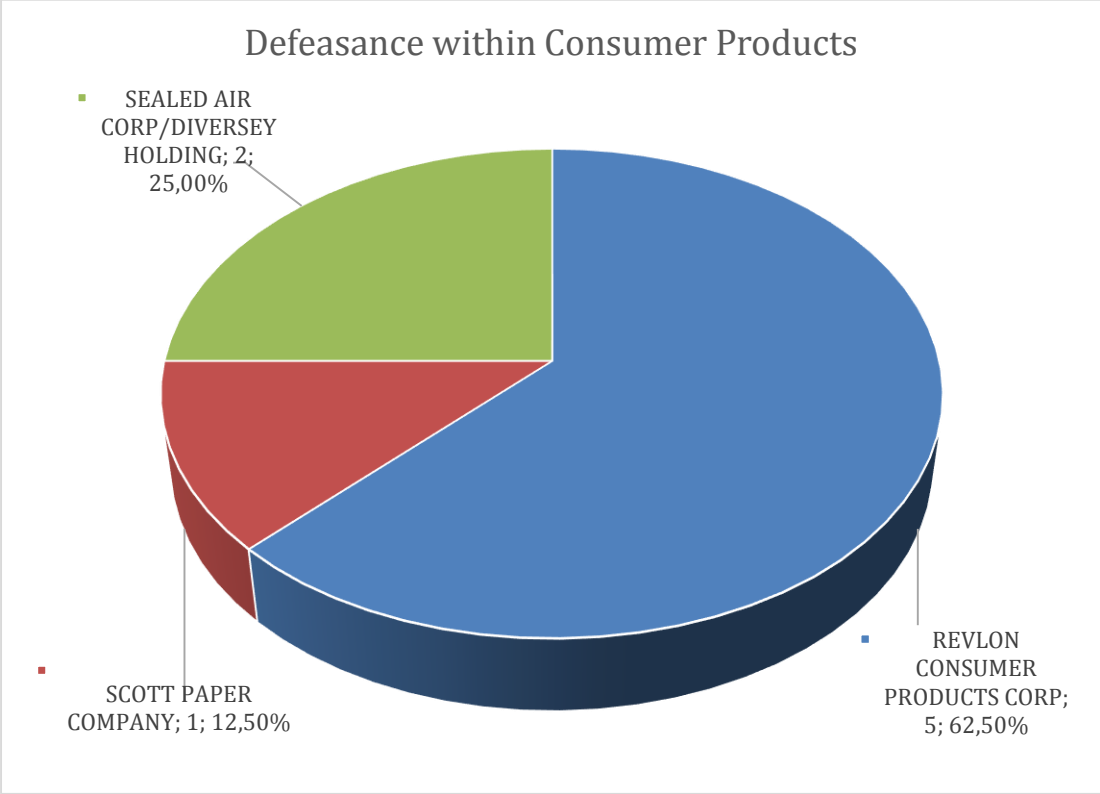

Figure 16: Firms in the consumer products industry that have exercised covenant defeasance options

The exercise of defeasance options in the consumer products industry is illustrated in figure 16. The company Revlon Incorporated dominates the defeasance findings in this category. In the category, the company represents 5 of 8 observations (62.5%) of the category and 12.5% of our total findings.

Revlon Incorporated is an American company listed on the NYSE in New York. More specifically, it produces cosmetics, fragrances, skin and personal care products. It has revenues of $ 1.49 BN, 6,500 employees, a market capitalization of $ 1.79 BN, and has historically been a financially healthy company (Revlon.com, 2014).

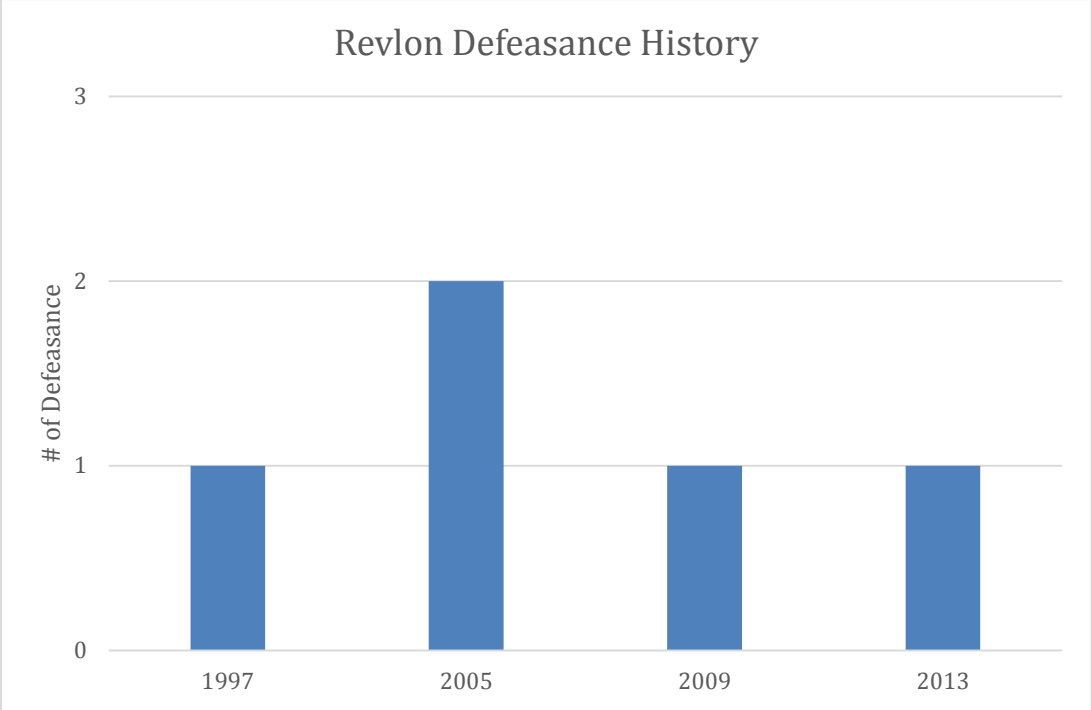

Figure 17: Revlon Incorporated defeasance history

Since Revlon has a history of exercising defeasance options, inquiries where done into when these exercises were made. Figure 17 shows that the five observations of covenant defeasance are spread across a 16-year time period, where the double observation in 2005 is linked to a refinancing where two bonds were involved. The 1997 observation is linked to a merger with Cosmetic Center Company (Baltimore Sun, 1996). For the 2013 and 2009 observations, significant corporate events could not be found.

Company	No. of Defeasance	Corporate Event	Date
PRICE COMMUNICATIONS WIRELESS	2	Acquisition	16.08.2002
THE RESTAURANT COMPANY/PERKINS	2	None found	01.08.2005
SEALED AIR CORP	2	Acquisition	02.12.2011

Table 5: Overview showing companies with more than one covenant defeasance option, reason for defeasance and defeasance date.

In addition to Revlon Incorporated, table 5 shows that there are three additional companies that have had more than one defeasance observation. While Revlon Incorporated has its defeasances spread over a period of time, Price Communication, Perkins (The Restaurant Company) and Sealed Air Corporation's defeasances are undertaken on more than one bond at the same date. Price Communication Wireless and Sealed Air Corporation did this due to acquisitions. Due to the fact that these two companies are associated with the same defeasance events, they cannot be considered independent.

With the exception of casinos and gaming, the industry of a company does not seem to affect the exercise of covenant defeasance options. The casinos and gaming industry is exposed to risks such as the issue and renewal of gaming licenses. Indeed, Aztar Corporations gaming division declared bankruptcy after failing to obtain such a license in 2010. However, Legg and Tang (2010) show how the casino industry historically has experienced low sensitivity to economic downturns, with a revenue growth of 3.1% during the recession of 2001. As most of the defeasance exercise observations are located around this period, it is difficult to argue that investors would see this industry as particularly risky. In addition, it cannot be determined that casinos and gaming are overrepresented, due to the small sample size. We therefore cannot conclude that industries exposed to high uncertainty are any more likely to exercise a covenant defeasance option.

5.5 Prediction 4: Callability is a substitute for covenant defeasance

According to our model, it is possible that callability will negatively affect the probability of defeasance exercise, as issuers can call the bond and thereby remove

all covenants. The bondholders cannot resist this method of covenant removal, since the decision to exercise the option lies wholly with the bond issuer. This eliminates the potential hold-up problems that might occur in repurchase offers.

Opposing the above view, Bienz et al (2013) point out that a large number of the callable bonds are issued with a make-whole premium. Of the defeasible bonds, 41.8% include such a premium. Bondholders also incur income tax on the proceeds of the call. The defeasance option does not expose the investor to reinvestment risk. Finding a new investment opportunity might not be attractive for the bondholder, especially in a low interest rate scenario where calling might be more beneficial over defeasance for the bond issuer. In contrast, a defeased bond exactly replicates the expected cash flow of the bond without risk of default.

In the regression, we investigate the relationship between a bond including a call option given by the independent variable *Callable*, and the exercise of covenant defeasance options. The intuition is that bond issuers who have the ability to call a bond may be less likely to exercise the covenant defeasance option, as they prefer to effect the call option instead. The relationship between exercising a call and exercising a covenant defeasance option is not investigated, as a bond cannot be terminated in more than one way. The regression output concerning the call option is outlined in table 6.

Specification Number	(1)	(2)	(3)	(4)	(5)	(6)
Data set	All bonds			Only bonds with defeasance options		
Callable	-0.553* (0.284)	-0.336 (0.301)	-0.336 (0.289)	-0.424 (0.299)	-0.291 (0.314)	-0.291 (0.301)
Covenant count	N	Y	Y	N	Y	Y
Robust standard errors	N	N	Y	N	N	Y
Only defeasible bonds	N	N	N	Y	Y	Y

Table 6: The regression output on the variable indicating callability using Is Defeased as dependent variable

The focus of the regression output is mainly on specifications (4), (5) and (6) in table 6, which are only conducted on bonds that have a defeasance option. This is because the effect on callability on the exercise of defeasance options is only relevant when the bond issuer has the choice between both options. Bond issuers who have not included a defeasance option may be forced to exercise their call option to remove covenants, even if they would have preferred exercising a covenant defeasance option. As we wish to investigate the exercise and not the inclusion of the defeasance option, this can potentially skew the results. Specifications (1) and (4) omit the variable *All covenants,* while specifications (3) and (6) use heteroskedasticity-consistent standard errors. The regression output concerning the call option is outlined in table 6.

Table 6 shows that the variable designating if the bond has a call option has low significant impact on the likelihood of exercising a covenant defeasance option. It is significant on the 90% level in specification (1) that excludes number of covenants and includes bonds that do not contain a defeasance option. In all other specifications, the variable is not significant. It is possible that the significance in specification (1) is due to an omitted variable bias, as the variable *Tender or Exchange Offer* is not included in the model. The variable is also not significant when the included bonds are reduced to only those that do not include a covenant defeasance option. This is important, as the callability variable is only interesting to us when the bondholder has a choice between calling and exercising a covenant defeasance option. When given such a choice, we find no indications that bond issuers who have the option to call are less likely to exercise a covenant defeasance than those who do not have this option. This supports the theory presented by Bienz et al (2013) that call options do not substitute covenant defeasance, due to refinancing risk and make-whole premiums of call options. The lack of significance does not disprove that there is exists a relationship between callability and covenant defeasance exercise.

5.6 Prediction 5: Defeased bonds contain more covenants

Our theory is that companies that are more restricted by covenants will have more incentive to exercise a covenant defeasance option. This is because a company with

many restrictive covenants is more likely to encounter situations where the covenants inhibit value-adding action, and might have to exercise the covenant defeasance option.

Bienz et al (2013) have shown that the inclusion of covenant defeasance options is positively associated with the number of covenants in a bond. The bond issuers are therefore willing to accept more restrictive covenants, because they have the option to remove them. This can lead to a potential bias, as any significant relationship between defeasance exercise and number of covenants, might be due to the bonds being defeasible instead. Specifications (5) and (6) in table 7 therefore include only defeasible bonds in the regression dataset, and we focus on the results of these three specifications.

In the regression output outlined in table 7, the independent variable *All covenants* is the number of restrictive covenants in each bond. This is used as a proxy for the degree of restriction. Note that specifications (1) and (4) are omitted as they do not include the *All covenants* variable. Specifications (3) and (6) use heteroskedasticity-robust standard errors.

Specification Number	(2)	(3)	(5)	(6)
Data set	All bonds		Only bonds with defeasance options	
All Covenants	0.0468*** (0.0169)	0.0468*** (0.0108)	0.0307* (0.0182)	0.0307*** (0.0114)
Robust standard errors	N	Y	N	Y
Only defeasible bonds	N	N	Y	Y

Table 7: The regression output on the variable measuring covenant restriction using Is Defeased as dependent variable

We predict that companies that are more restricted will be more inclined to exercise their covenants defeasance options. We therefore expect a positive relationship between the number of covenants and the exercise of defeasance options. In all

specifications seen in table 7, the regression coefficient is positive, which is in accordance with our expectations.

There is a marked decrease in the explanatory power of the *All covenants* variable when the dataset is restricted to only include bonds with a defeasance option. This is in accordance with the positive relationship between the number of covenants and inclusion of covenant defeasance options documented by Bienz et al (2013). Bonds with fewer covenants are less likely to include defeasance options, and the average number of covenants will therefore be higher when the bonds without a defeasance option are excluded. The average number of restrictive covenants is 8.06 for all bonds and 9.45 for the bonds that contain a defeasance option. This multicollinearity can also be seen in the correlation table in table 2, where the correlation coefficient is 0.42.

In specifications (5) and (6), where the dataset only includes defeasible bonds, the regression coefficients in table 7 are still significant at the 90% and 99% level respectively. This indicates that among bonds that include a defeasance option, those with a higher number of covenants are more likely to exercise their covenant defeasance options.

In conclusion, the regressions indicate that there is a strong positive correlation between the number of restrictive covenants in a bond, and covenant defeasance exercise. This is in accordance with our assumption that bonds that are more restricted will have more reason to exercise a covenant defeasance than less restricted bonds. It is also interesting that the number of covenants is still significant when only bonds with defeasance options are included. This eliminates the potential bias created by the correlation between the number of covenants and the inclusion of a defeasance option documented by Bienz et al (2013).

5.7 Prediction 6: Defeasance is exercised in conjuncture with major corporate events

We predict that the need to remove restrictive covenants arises when the company is undergoing major corporate actions. This is because the restrictive covenants will limit the company's freedom to act, and a major value-adding action is needed to outweigh the cost associated with covenant defeasance. Such actions may be mergers, acquisitions and refinancing.

The analysis is conducted only on the defeased bonds, and is not incorporated into the regression analysis. This is because corporate events are somewhat difficult to quantify. No dataset available to the authors lists such actions. Attempts at using proxies such as change in debt was unsuccessful, as it proved to have weak correlation to manually gathered information on corporate events. The types of events that may be restricted by covenants are numerous, and does not necessarily give the same effects on potential proxies. As such, the analysis is done only on the defeased bonds, and does not include an associated regression.

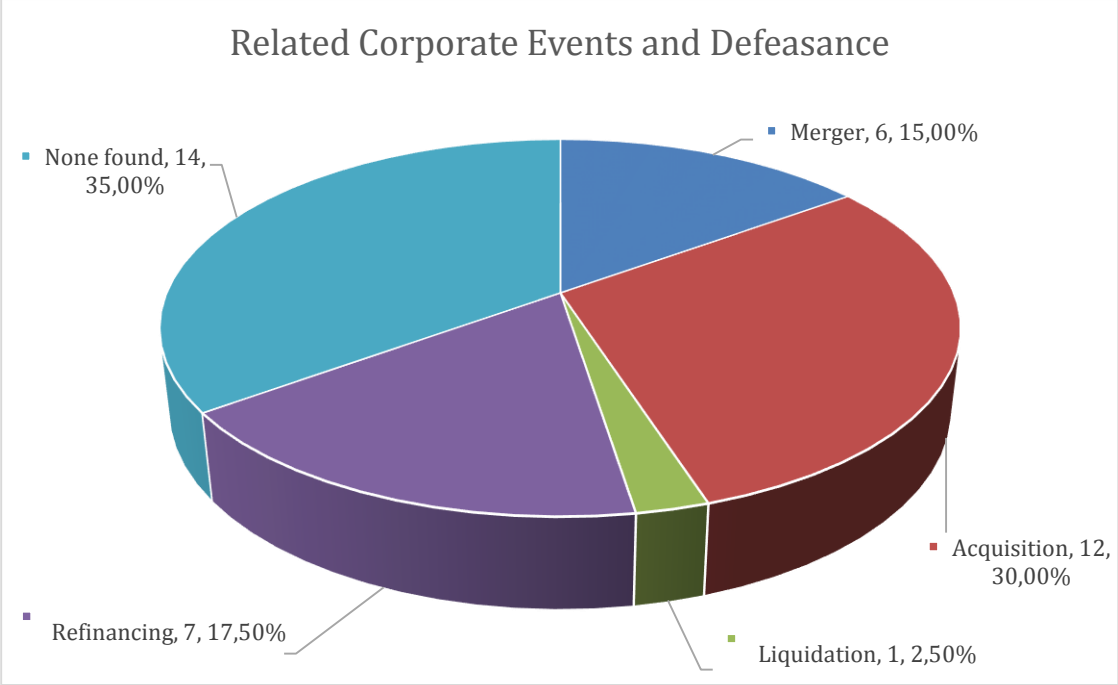

Figure 18: Significant corporate events among companies that have exercised a defeasance option

Figure 18 shows related corporate events for the total sample of defeasance option exercise. We find that 65% of the defeasance exercises are linked to a corporate event like an acquisition, merger or refinancing. This link was often explicit, where the company stated that the defeasance exercise took place to facilitate the corporate action. 45% of the covenant defeasance options were exercised prior to a merger or acquisition, 17.5% were related to a refinancing of a company and one observation where in connection with a liquidation. In 35% of the exercises, no significant corporate event was found.

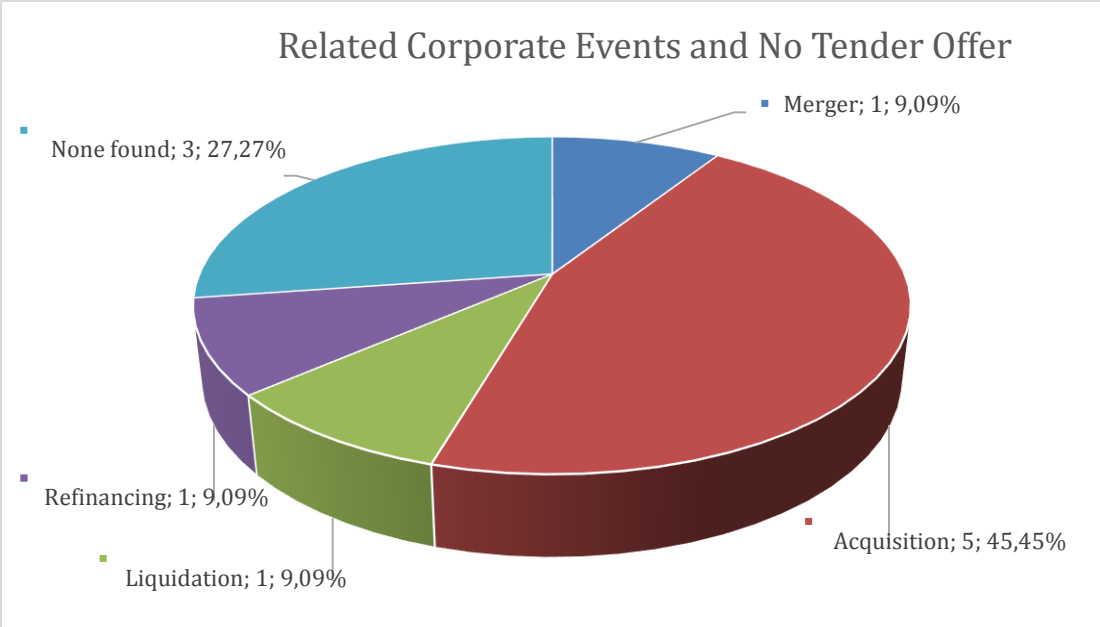

Figure 19: Corporate events in the sample that did not tender the defeased bond

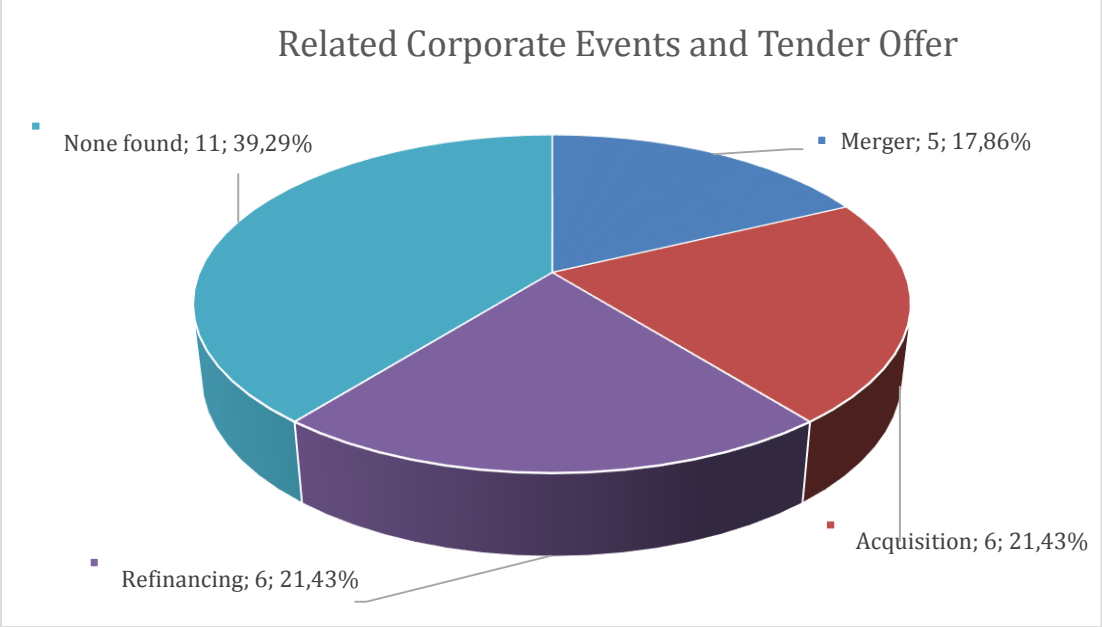

Figure 20: Corporate events in companies that attempted tender before exercising the defeasance option

Figures 19 and 20 show the related corporate events of the companies that did and did not tender their bonds before defeasance exercise. The companies that did not tender were more involved in mergers and acquisitions (55% of the sample) than the companies that did tender (40% of the sample). In addition, the companies that did not tender had only 1 observation of refinancing vs. 6 in the companies that did tender.

In the case of the merger between Price Communication and Verizon in 2002, a defeasance option is exercised so that the company can circumvent the covenant "Change of Control". The bond agreement explicitly state that the bond should be redeemed immediately one day following the closing date of the transaction (Price Communications Cellular Holdings, 2000). In the same filing, it is stated that Price Communications will exercise a covenant defeasance of the outstanding bonds provided that the merger goes through and that the acquirer, Verizon, provided the necessary cash agreed upon to exercise such a defeasance.

In 1996, Revlon acquired the company Prestige Fragrance & Cosmetics (Baltimore Sun, 1996). This required the issuance of more debt, which was restricted by covenants in

existing bonds. The company thereby defeased the existing bonds and issued new bonds for Revlon's own financing needs and for the acquisition.

Covenant defeasance exercise seems to be more likely to happen in the events of major corporate events that breaches or potentially breaches the covenants of a company's outstanding bond issues. The major found corporate events are mergers, acquisitions or refinancing. This is according to our expectations.

5.8 Limitations of the Analysis

The dataset of defeased bonds contains only 40 records of defeasance exercise. This creates problems when the data is compared to the FISD dataset. Only 21 of the 40 bonds that were found to be defeased was found in the FISD database, even after attempted manual matching. We have not found any indication that there are any variables that affect what bonds cannot be found in the FISD. The existence of such a bias cannot be disproven. This can skew the results of the regression due to attributes of the bonds that have not been matched. A larger dataset of exercised defeasance options would increase the robustness of the regression. However, we have high confidence that most instances of covenant defeasance exercise have been identified in our dataset. As mentioned in section 4.4, alternative methods of identifying covenant defeasance yielded no new instances of covenant defeasance exercise compared to using the self-developed search program.

The low number of defeasance exercise has also precluded the inclusion of category variables, as there are too few observations in each category. An attempt was made to include dummy variables for industry, but there were no more than four observations in each industry variable. This was not acceptable as the number of firms in each industry was more important to the statistical significance, than the number of firms that had defeased. The decision to not include the industry variables can give omitted variable bias. This can increase the significance of the variables that have been included in the regression.

Some variables where omitted from the regression due to missing observations. Bienz et al (2013) show how bond issuer fundamentals such as the fixed asset ratio can affect the inclusion of covenant defeasance options. It is possible that such variables could be significant when examining defeasance exercise. When attempting to include fixed assets from COMPUSTAT, missing values further reduced the number of defeasance observations in the regressions to 13. This loss of data fidelity was not considered as justified by the variable potential significance. The companies that could be found in COMPUSTAT appeared to be larger companies, which could create potential bias towards large companies in regressions. A more robust dataset on company fundamentals could improve the analysis, but this was not available to the authors.

We must therefore admit the possibility of omitted variable bias in the regression output, due to missing variables concerning bond issuer fundamentals, and omitting category variables.

6 Conclusion

In this thesis, we illustrate the composition of covenant defeasance exercise in the period 1993 to mid-2014. With the use of a self-written computer search program, we were able to search through 1.4 million SEC-filings and make a unique dataset of exercised covenant defeasance options. By linking our dataset to FISD and by performing empirical analysis, we were able to gain insights into why and how covenant defeasance options are exercised.

In the examined time period, we find 40 exercised covenant defeasance options. These are spread across a wide variety of industries and are quite evenly distributed through the time period. A higher number of exercises than we expected are observed in the casinos and gaming industry, but the number of observations is not high enough to conclusively show a difference between this and other industries.

Covenant defeasance options are usually exercised in conjunction with a tender offer where the bond issuer attempts to buy back the bond before exercising the defeasance option. This indicates that the defeasance option might be costlier to exercise than buying back the bonds in the market. In a majority of the cases, exercises are most often performed prior or after a major corporate event like mergers, acquisitions or refinancing.

We find little indication that call options substitute for defeasance exercise, as bonds with the option to call does not exercise their defeasance options significantly less often. This supports previous literature on the subject by Bienz et al (2013) showing that call options does not substitute defeasance options.

Through the creation of a unique dataset, this thesis gives insights into the exercises of covenants defeasance options. We also document specific connected traits that can help understand the use of these options. Even though covenant defeasance options are often included in bond issues, there is little information about covenant defeasance exercise. Familiarity with these options to both holders and the affected

party might be limited. The insights in this thesis can be of use to holders of the option or potential affected parties, to better understand the potential contractual consequences of entering such agreements.

Appendix

Table 1: Summary Statistics of Regression Variables

We present summary statistics, including the mean, the standard deviation, the minimum and the maximum for a sample of 10604 US corporate bonds. The information is from the Fixed Income Securities Database, with the exception of the *Is Defeased* variable, which is from the authors' original research. *Tender or Exchange Offer, Callable, Has Defeasance Option* and *Is Defeased* are dummy variables, and their mean is therefore given as percentages to improve readability. The included variables are whether the covenant defeasance option of the bond is exercised, whether the bond has had at least one tender or exchange offer made, whether the bond is callable, the number of restrictive covenants on the bond, and whether the bond has a defeasance option included.

Variable	N	Mean	Std. Dev.	Min	Max
Is Defeased	10604	0,19 %	0,043	0	1
Tender or Exchange Offer	10604	12,07 %	0,326	0	1
Callable	10604	27,24 %	0,445	0	1
All Covenants	10582	8,06	4,868	0	26
Has Defeasance Option	10604	67,80 %	0,467	0	1

Table 2: Covariance Matrix

This table shows the statistical relationship between the dependent and independent variables. It is calculated using the full FISD dataset of all 10607 corporate US-owned bonds. The variable *Has Defeasance Option* is included although it is not an independent variable. It has moderate correlation with the independent variable *All Covenants*. The regression specifications (4) (5) and (6) are for this reason done on only bonds that has a defeasance option.

	Is Defeased	*Tender or Exchange Offer*	*Callable*	*All Covenants*	*Has Defeasance Option*
Is Defeased	1,00				
Tender or Exchange Offer	0,03	1,00			
Callable	-0,02	-0,06	1,00		
All Covenants	0,04	0,17	-0,36	1,00	
Has Defeasance Option	0,02	0,11	-0,23	0,42	1,00

Table 3: Regression Outputs

We run Probit regressions with *Is Defeased* as the dependent variable to test predictions 1-4. *Is Defeased* takes value one when the defeasance option is found to have been exercised, and zero otherwise. We include the variables that drive the exercise of defeasance option as hypothesized: The number of covenants, if the bond is callable and if a tender or exchange offer has been made on the bond. We compare specifications 1-3 with 4-6 to control for the relationship between the number of covenants and inclusion of defeasance options. Specifications 4-6 have regressions made only on defeasible bonds. Specifications (1) and (4) omit the variable for number of covenants, to control the sensitivity of the remaining variables to under-specification. Specifications (3) and (6) are done using robust standard errors to control for potential multicollinearity indicated in the correlation matrix. Standard errors in parentheses *** p<0.01, ** p<0.05, * p<0.1

Specification Number	(1)	(2)	(3)	(4)	(5)	(6)
Dependent Variable	*Is Defeased*	*Is Defeased*	*Is Defeased*	*Is Defeased*	*Is Defeased*	*Is Defeased*
Tender or Exchange Offer	0.421***	0.305*	0.305*	0.341**	0.307*	0.307*
	(0.159)	(0.170)	(0.177)	(0.171)	(0.173)	(0.179)
Callable	-0.553*	-0.336	-0.336	-0.424	-0.291	-0.291
	(0.284)	(0.301)	(0.289)	(0.299)	(0.314)	(0.301)
All Covenants		0.0468***	0.0468***		0.0307*	0.0307***
		(0.0169)	(0.0108)		(0.0182)	(0.0114)
Constant	-2.904***	-4.109***	-4.109***	-2.830***	-3.845***	-3.845***
	(0.0884)	(0.548)	(0.602)	(0.0943)	(0.571)	(0.626)
Observations	10 604	10 570	10 570	7 190	7 182	7 182
Pseudo R-squared	0,0446	0,0717	0,0717	0,0268	0,0443	0,0443
Covenant count	N	Y	Y	N	Y	Y
Robust standard errors	N	N	Y	N	N	Y
Only defeasible bonds	N	N	N	Y	Y	Y

Code for Downloading Index Files

This code relates to section 4.1.4.1, and contains all the steps that facilitate the downloading of index files from the SEC webpages.

```
// Method to only save index files for the last quarter in the database and after. Used to quickly update the records to present day, but will not legacy control.
        private void GetQuickIndexFiles(BackgroundWorker bw)
        {
            //Deletes all index files in the index folder as they might be incomplete/outdated
            string Savepath = (@AppDir + ":\\EDGAR\\EDGARINDEX\\");
            System.IO.DirectoryInfo directory = new System.IO.DirectoryInfo(Savepath);
            directory.Empty();
            //Gets the current year and quarter
            Int32 ThisYear = DateTime.Now.Year;
            Int32 ThisQuarter = Research.GetQuarter(DateTime.Now);
            //Gets the newest date currently in the database
            DateTime maxdate = getmaxdate();
            //Gets the quarter of the newest date
            Int32 MaxQuarter = Research.GetQuarter(maxdate);
            //Creates year-items for between today and last date in database
            for (int y = maxdate.Year; y <= ThisYear; y++)
                //If the year to be downloaded is this year, we need to check which quarters to download
                 if (y == ThisYear)
                     if (ThisQuarter >= MaxQuarter)
                     {
                    for (int q = Research.GetQuarter(maxdate); q <= ThisQuarter; q++)
                        DownloadIndex(y, q,bw);
                     }
                     else
                     {
                         for (int q = 1; q <= ThisQuarter; q++)
                             DownloadIndex(y, q, bw);
                     }
```

```
                else
                    for (int q = 1; q <= 4; q++)
                        DownloadIndex(y, q,bw);
            bw.ReportProgress(1, "Done downloading index files");

        }
```

Code for Parsing Index Files to Memory

This code relates to section 4.1.4.2 and contains the code necessary to read the index files into memory.

```
        public void ReadFile(bool Quicksearch, BackgroundWorker bw)
        {



            // Sets the adress of the index file folder
            string sti = @AppDir+":/Edgar/EDGARINDEX/";
            //creates a string array of all the filenames in the
index file folder
            string[] F = Directory.GetFiles(sti);
            //Finds the newest record in the database.
            DateTime maxdate =  getmaxdate();
            //Goes through each index file in the folder
            foreach (string Filename in F)
            {
                bw.ReportProgress(1,"Parsing file " +Filename);
                //Initiates the quarter and year variables for
helping the saver
                int lookupyear = 0;
                int lookupquarter = 0;
                //Creates a new document feed list to contain the
records from one index file
                List<DocumentInstance>    _DocumentFeed    =    new
List<DocumentInstance>();
                //Opens the index file in the reader, that reads the
file one line at a time
```

```
                using (System.IO.StreamReader IndexText = new
System.IO.StreamReader(Filename))
                {

                    Boolean Nyttformat = false;
                    DateTime Result;
                    String IndexLine = String.Empty;
                    Int32 cik;
                    IndexLine = IndexText.ReadLine();
                    //Goes through the file untill it finds a line
of dashes, indicating the end of the header and start of the data.
                    while (IndexLine.IndexOf("-----------------------
--------------------") == -1)
                    {
                        // There are two main formats of the index
files. The old format had set width for each data item, while the
new one separates the items using | as a delimiter.
                        //If the items are separated by | instead of
set with spaces, we set the nyttformat bool to true
                        if (IndexLine.IndexOf("CIK|") == -1)
                        { }
                        else
                            Nyttformat = true;

                        IndexLine = IndexText.ReadLine();

                    }
                    //Reads an extra line down to move from the
dashed line to the first record line.
                    IndexLine = IndexText.ReadLine();
                    // Read each line untill there are no more lines
in the file
                    while (IndexLine != null)
                    {
                        //Declares a single docment instance object
to hold a single record
                        var _e = new DocumentInstance();
                        //If the file is of the new format as
determined above, we split the line into a string array on the |
symbol
```

```
                        if (Nyttformat)
                        {
                            //Splits the line
                            String[]          IndexItems          =
IndexLine.Split('|');
                            //Insert the relevant data items into
the relevant object class entities.
                            _e.CompanyName = IndexItems[1];
                            _e.FormType = IndexItems[2];
                            //Uses tryparse in case the cik is
corrpted(ie. not just numbers)
                            if  (Int32.TryParse(IndexItems[0],  out
cik))
                                _e.CIK = cik;
                            string dateitem = IndexItems[3];
                            // Tries to parse the publication date
using a specified format. If it fails to report a date, the
pblicationdate will b null. This item will be subject to change
should the SEC change the date format.
                            if    (DateTime.TryParseExact(dateitem,
"yyyy-MM-dd", CultureInfo.InvariantCulture, DateTimeStyles.None, out
Result))
                                _e.PublicationDate = Result;
                            _e.FileLink = IndexItems[4];

                        }
                            // If the index file is of the old
format, the items are separated by set width. This is a less robust
system that also limits possible company name length to 61 characters
                            // This method should never be invoked
if one only uses freshly downloaded index files, as even old index
files have been updated to the new format on EDGAR.
                            //I left it in just in case.

                        else
                        {
                            // The first number in the substring
method represents the start index position in the string, the second
specifies the length of the substring that should be retrived after
the index position.
```

```
                            // The .trim() method removes any
leading and trailing blank spaces from the retrived string.
                            _e.CompanyName = IndexLine.Substring(0,
61).Trim();
                            _e.FormType = IndexLine.Substring(61,
10).Trim();
                            //Tries to parse the CIK into an integer
                            if
(Int32.TryParse(IndexLine.Substring(74, 10).Trim(), out cik))
                                _e.CIK                              =
Convert.ToInt32(IndexLine.Substring(74, 10).Trim());
                            // The set width format uses two
different date formats. This determines which to use.
                            if              (IndexLine.Substring(84,
10).IndexOf("-") == -1)
                            {
                                string          dateitem          =
IndexLine.Substring(84, 10).Trim();
                                if
(DateTime.TryParseExact(dateitem,                        "yyyyMMdd",
CultureInfo.InvariantCulture, DateTimeStyles.None, out Result))
                                    _e.PublicationDate = Result;
                            }
                            else
                            {
                                string          dateitem          =
IndexLine.Substring(84, 14).Trim();
                                if
(DateTime.TryParseExact(dateitem,                      "yyyy-MM-dd",
CultureInfo.InvariantCulture, DateTimeStyles.None, out Result))
                                    _e.PublicationDate = Result;
                            }
                            _e.FileLink = IndexLine.Substring(98,
50).Trim();

                        }
                        //Sets the lookupyear to the year of the item
parsed. Thus, the last item parsed will determine what year the index
file is for.
                            lookupyear = _e.PublicationDate.Year;
```

```
                            lookupquarter    =    Research.GetQuarter
(_e.PublicationDate);
                        //  If  the  quick  search  method  has  been
invoked, only files with a publication date the same or later than
the newest in the database will be added.
                        // Otherwise all files will be parsed
                            if (Quicksearch == true)
                            {
                                if (_e.PublicationDate >= maxdate)
                                    _DocumentFeed.Add(_e);
                            }
                            else
                                _DocumentFeed.Add(_e);
                        //Readies the next line in the document to
be read
                            IndexLine = IndexText.ReadLine();
                        //Loops to read the next line in the document
                    }
                }
                bw.ReportProgress(1,"Found " + _DocumentFeed.Count);
                //Invokes  the  saver  to  save  the  items  parsed  from
the index file
                 Saver(_DocumentFeed,lookupyear,lookupquarter,bw);
                //Loops to the next index files to be parsed
            }

            bw.ReportProgress(1, "Done parsing");
```

Code for Saving Index Information to the Database

This code relates to section 4.1.4.3 and contains the code necessary to save data to the database

```
public  static  void  Saver(  List<DocumentInstance>  lur,int  lookupyear,int
lookupquarter,BackgroundWorker bw)
        {
            //If no items are parsed, we quit this method
            if (lur.Count <= 0)
               return;
```

```
            //The information needed to conect to the database. Appdir
contains the driveletter if the drive the program is run from
            string                cs                =                "Data
Source="+AppDir+":\\EDGAR\\edgar_INDEX.sqlite;Version=3;";
            Int64 Maxindex = 0;



            //Sets the connection to the database
            SQLiteConnection conn = new SQLiteConnection(cs);
            bw.ReportProgress(1,"Saving    records    for    year    "+
lookupyear.ToString() + " quarter "+lookupquarter.ToString());
            {
                //Opens the database connection
                conn.Open();
                //Declares a transaction on the connection. This ensures
no data is commited to the database untill all items are inserted without
errors. This ensures partial insertions are not made
                //when an error is encountered. The data is stored only
when the transaction is commited.
                var transaction = conn.BeginTransaction();
                //try-catch loop to handle errors on insertion
                try
                {
                    // new list to contain the existing records from the
database. This data is used to check if the data to be inserted exists in
the database allready.
                    List<DocumentInstance>         h         =         new
List<DocumentInstance>();
                    var cmd = new SQLiteCommand();
                    cmd.Connection = conn;
                    cmd.Transaction = transaction;
                    var cmd2 = new SQLiteCommand();
                    cmd2.Connection = conn;

                    //Sql to find the highest index id allready in the
database. This is used to create new indexid itms(primary key)
                    cmd2.CommandText    =    "Select    MAX(indexid)    from
raportindex";
                    var Maxid = cmd2.ExecuteReader();
                    while (Maxid.Read())
```

```
                    {
                        Maxindex = Convert.ToInt64(Maxid[0]);
                    }
                    //We add one to the index id retrived to prepare for
inserion
                    Maxindex++;

                    cmd2.Dispose();
                    //Sql to find existing items in the database from the
year and quarter that the parsed file is for. We check the new data agianst
this list to avoid duplicate entries
                    cmd2.CommandText = "Select Filename from raportindex
where  strftime('%Y',datefiled)  =  '"  +  lookupyear  +  "'  and
(((cast(strftime('%m',  datefiled)  as  integer)  -1  )  /  3)+1)  =  "  +
lookupquarter+"";
                   //Execute the get existing items command
                    var p = cmd2.ExecuteReader();
                    //Reads  the  found  records  from  the  database  result
variable  to  the  document  instance  list  in  local  memory,  so  that  we  can
dispose the command.
                    while (p.Read())
                    {
                    var _q = new DocumentInstance();

                       _q.FileLink= (""+p[0]);
                       h.Add(_q);
                    }
                    //Dispose  the  command,  so  that  we  are  ready  for
insertion.
                    cmd2.Dispose();



                    // This method compares the filelink atribute in the
list of existing database items with the filelinks in the parsed data. We
use filelink as it is the only candidate key in the
                    //parsed data. There can be multiple forms of the same
type by the same company on the same time. Maching on non-indexed strings
like this is verry cpu and time itensive however.
                    //The method returns the list object wic will contain
all records where the parsed filelink was not found in the database
```

```
                        var list = from g  in lur
                                   where !(from o in h
                                           select o.FileLink)
                                           .Contains(g.FileLink)
                                   select g;
                    //If no records that didn't allready exist where found,
we exit the method to parse the next text file
                        if (list.Count() < 1)
                            return;
                    //Saves each item to the database.
                    foreach (var Item in list)
                    {



                        cmd.Connection = conn;
                        cmd.Transaction = transaction;
                        //Parameterize all atributes of the item. This is
slightly overkill on this program from a security standpoint(as the database
is completely open) but it is still good practice
                        AddParameter(cmd.Parameters, "@IndexID", Maxindex,
DbType.Int64);
                        AddParameter(cmd.Parameters,       "@CompanyName",
Item.CompanyName, DbType.String, 62);
                        AddParameter(cmd.Parameters,          "@FormType",
Item.FormType, DbType.String, 10);
                        AddParameter(cmd.Parameters,   "@CIK",   Item.CIK,
DbType.Int64);
                        AddParameter(cmd.Parameters,         "@DateFiled",
Item.PublicationDate, DbType.DateTime);
                        AddParameter(cmd.Parameters,          "@FileName",
Item.FileLink, DbType.String, 50);
                        //Define the insertion string
                        cmd.CommandText   =   "Insert   into   RaportIndex
(IndexID,CompanyName,     FormType,     CIK,DateFiled,Filename)     VALUES
(@IndexID,@CompanyName,@Formtype,@CIK,@DateFiled,@FileName)";



                        //Add item to database(it is not completly saved
untill the transaction is committed)
```

```
                        cmd.ExecuteNonQuery();

                    // Add one to the index id to prepare for the next
insertion
                        Maxindex++;
                    }
                    //If all items in the list are saved correctly, this
method commits the changes to the database.
                    transaction.Commit();
                    //Closes the connection
                    conn.Close();
                    }
                // If any errors are encounterd douring insertion, this
method catches them
                catch (Exception)
                {
                    bw.ReportProgress(1, "Error, rolling back");
                    //We roll back any changes made to the database in this
method, meaning any records that was inserted before the error was thrown
is not inserted.
                    transaction.Rollback();
                    bw.ReportProgress(1, "Finished rolling back");

                }
                finally
                {
                    if (conn.State == ConnectionState.Open)
                        conn.Close();
                }

            }


        }
```

Code for Downloading SEC Forms to Local Storage

This code relates to section 4.1.4.4 and contains the code used to download the SEC forms themselves to the local hard drive.

```
public void GetMasterList(string formtype)
        {
            string aarSQL = "Select * From formyears;";
            string              cs              =              "Data
Source="+AppDir+":\\EDGAR\\edgar_INDEX.sqlite;Version=3;";
            string GetYear;
            System.IO.StreamWriter       ErrorList       =       new
System.IO.StreamWriter(AppDir+":\\EDGAR\\errorlist.TXT");
            SQLiteConnection conn = new SQLiteConnection(cs);
            conn.Open();
            var cmd = conn.CreateCommand();
            cmd.CommandText = aarSQL;
            bw.ReportProgress(1,"Getting   distinct   years   from
database");
            var aar = cmd.ExecuteReader();
            List<String> aaar = new List<String>();
            while (aar.Read())
            {
                aaar.Add("" + aar[0]);

            }
            conn.Close();
            foreach (string ar in aaar)
            {

                SQLiteConnection iconn = new SQLiteConnection(cs);
                iconn.Open();
                GetYear = "Select IndexID, FileName, Companyname,
Datefiled from raportindex where strftime('%Y',Datefiled)='" + ar +
"' and FormType like '" + formtype + "'";
                var icmd = iconn.CreateCommand();
                string savepath = @AppDir+":\\EDGAR\\" + formtype +
"\\" + ar + "\\";
                if (!Directory.Exists(savepath))
                {
                    Directory.CreateDirectory(savepath);
                }

                icmd.CommandText = GetYear;
```

```
                bw.ReportProgress(1,"Getting " + formtype + "s for
year " + ar);
                var f = icmd.ExecuteReader();
                List<DocumentInstance>        g        =        new
List<DocumentInstance>();
                while (f.Read())
                {
                    var _e = new DocumentInstance();
                    _e.IndexID =Convert.ToInt32(f[0]);
                    _e.FileLink = "" + f[1];
                    _e.CompanyName = "" + f[2];
                    _e.PublicationDate = Convert.ToDateTime(f[3]);
                    g.Add(_e);
                }
                iconn.Close();
                WebClient Request = new WebClient();
                //Request.Credentials              =              new
NetworkCredential("anonymous", "nhhpost@gmail.com");
                foreach (var d in g)
                {
                    // Television recording is beginning. Enable
away mode and prevent
                    // the sleep idle time-out.


                OmIgjen:
                    SetThreadExecutionState(
                        ES_CONTINUOUS |
                        ES_SYSTEM_REQUIRED |
                        ES_AWAYMODE_REQUIRED);
                    if     (DateTime.UtcNow.Hour     >=     02     &&
DateTime.UtcNow.Hour < 24)
                    {



                        string                 url                 =
"http://www.sec.gov:80/Archives/" + d.FileLink;
                        if    (!File.Exists(@AppDir+":\\EDGAR\\"    +
formtype + "\\" + ar + "\\" + d.IndexID.ToString() + ".txt"))
```

```
                        {
                            try
                            {
                                bw.ReportProgress(1, "Saving file "
+      d.IndexID      +      ".txt      PublicationDate      "      +
d.PublicationDate.ToShortDateString()    +    "    Company    "    +
d.CompanyName.ToString());
                                Console.WriteLine(" Saving file " +
d.IndexID + ".txt");
                                Request.DownloadFile(url,
@AppDir+":\\EDGAR\\"   +   formtype   +   "\\"   +   ar   +   "\\"   +
d.IndexID.ToString() + ".txt");

                            }
                            catch (WebException ex)
                            {
                                ErrorList.WriteLine(d.IndexID + ";"
+ d.FileLink + ";" + ex.InnerException);
                                bw.ReportProgress(1,"Error
downloading company" + d.IndexID.ToString());
                                StreamWriter      err      =      new
StreamWriter(@AppDir+ ":\\EDGAR\\" + formtype + "\\" + ar + "\\" +
d.IndexID.ToString() + ".txt");
                                err.WriteLine("Error Downloading");
                                err.Close();


                            }

                        }
                    }
                    else
                    {
                        bw.ReportProgress(1, "Current time within US
working hours, Download paused");
                        System.Threading.Thread.Sleep(60000);
                        goto OmIgjen;
                    }

                }
```

```
            }
                bw.ReportProgress(1, "Download complete");
                SetThreadExecutionState(ES_CONTINUOUS);
                ErrorList.Close();
        }

        private void cmdDoDownload_Click(object sender, EventArgs e)
        {
            string Formtype = cmbUpdate.Text;
            bw.WorkerReportsProgress = true;
            bw.WorkerSupportsCancellation = true;
            bw.DoWork += new DoWorkEventHandler(delegate(object o,
DoWorkEventArgs args)
            {
                BackgroundWorker b = o as BackgroundWorker;

                GetMasterList(Formtype);

                // report the progress in percent

                 });

                bw.ProgressChanged              +=              new
ProgressChangedEventHandler(
        delegate(object o, ProgressChangedEventArgs args)
        {
            string b = args.UserState as string;

            lblProgress.Text = b;




        });

bw.RunWorkerAsync();

        }
```

```
    }
}
```

Code for Searching Downloaded Forms for Specified Search String

This section relates to section 4.1.4.5 and contains the code that reads through the forms, and returns result to a data file.

```
public void GetMasterList(string formtype)
        {
            string aarSQL = "Select * From formyears;";
            string cs = "Data Source="+AppDir+":\\EDGAR\\edgar_INDEX.sqlite;Version=3;";
            string GetYear;
            System.IO.StreamWriter ErrorList = new System.IO.StreamWriter(AppDir+":\\EDGAR\\errorlist.TXT");
            SQLiteConnection conn = new SQLiteConnection(cs);
            conn.Open();
            var cmd = conn.CreateCommand();
            cmd.CommandText = aarSQL;
            bw.ReportProgress(1,"Getting distinct years from database");
            var aar = cmd.ExecuteReader();
            List<String> aaar = new List<String>();
            while (aar.Read())
            {
                aaar.Add("" + aar[0]);

            }
            conn.Close();
            foreach (string ar in aaar)
            {

                SQLiteConnection iconn = new SQLiteConnection(cs);
                iconn.Open();
                GetYear = "Select IndexID, FileName, Companyname, Datefiled from raportindex where strftime('%Y',Datefiled)='" + ar + "' and FormType like '" + formtype + "'";
                var icmd = iconn.CreateCommand();
```

```
                string savepath = @AppDir+":\\EDGAR\\" + formtype +
"\\" + ar + "\\";
                if (!Directory.Exists(savepath))
                {
                    Directory.CreateDirectory(savepath);
                }

                icmd.CommandText = GetYear;
                bw.ReportProgress(1,"Getting " + formtype + "s for
year " + ar);
                var f = icmd.ExecuteReader();
                List<DocumentInstance>        g        =        new
List<DocumentInstance>();
                while (f.Read())
                {
                    var _e = new DocumentInstance();
                    _e.IndexID =Convert.ToInt32(f[0]);
                    _e.FileLink = "" + f[1];
                    _e.CompanyName = "" + f[2];
                    _e.PublicationDate = Convert.ToDateTime(f[3]);
                    g.Add(_e);
                }
                iconn.Close();
                WebClient Request = new WebClient();
                //Request.Credentials             =             new
NetworkCredential("anonymous", "nhhpost@gmail.com");
                foreach (var d in g)
                {
                    // Television recording is beginning. Enable
away mode and prevent
                    // the sleep idle time-out.


                OmIgjen:
                    SetThreadExecutionState(
                        ES_CONTINUOUS |
                        ES_SYSTEM_REQUIRED |
                        ES_AWAYMODE_REQUIRED);
                    if     (DateTime.UtcNow.Hour     >=     02     &&
DateTime.UtcNow.Hour < 24)
```

```
                    {



                        string                  url                 =
"http://www.sec.gov:80/Archives/" + d.FileLink;
                        if    (!File.Exists(@AppDir+":\\EDGAR\\"    +
formtype + "\\" + ar + "\\" + d.IndexID.ToString() + ".txt"))
                        {
                            try
                            {
                                bw.ReportProgress(1, "Saving file "
+      d.IndexID      +      ".txt      PublicationDate      "      +
d.PublicationDate.ToShortDateString()    +    "    Company    "    +
d.CompanyName.ToString());
                                Console.WriteLine(" Saving file " +
d.IndexID + ".txt");
                                Request.DownloadFile(url,
@AppDir+":\\EDGAR\\"   +   formtype   +   "\\"   +   ar   +   "\\"   +
d.IndexID.ToString() + ".txt");

                            }
                            catch (WebException ex)
                            {
                                ErrorList.WriteLine(d.IndexID + ";"
+ d.FileLink + ";" + ex.InnerException);
                                bw.ReportProgress(1,"Error
downloading company" + d.IndexID.ToString());
                                StreamWriter      err      =      new
StreamWriter(@AppDir+ ":\\EDGAR\\" + formtype + "\\" + ar + "\\" +
d.IndexID.ToString() + ".txt");
                                err.WriteLine("Error Downloading");
                                err.Close();


                            }

                        }
                    }
                    else
```

```
                    {
                        bw.ReportProgress(1, "Current time within US
working hours, Download paused");
                        System.Threading.Thread.Sleep(60000);
                        goto OmIgjen;
                    }

                }


            }
                bw.ReportProgress(1, "Download complete");
                SetThreadExecutionState(ES_CONTINUOUS);
                ErrorList.Close();
        }

        private void cmdDoDownload_Click(object sender, EventArgs e)
        {
            string Formtype = cmbUpdate.Text;
            bw.WorkerReportsProgress = true;
            bw.WorkerSupportsCancellation = true;
            bw.DoWork += new DoWorkEventHandler(delegate(object o,
DoWorkEventArgs args)
            {
                BackgroundWorker b = o as BackgroundWorker;

                GetMasterList(Formtype);

                // report the progress in percent

                 });

                bw.ProgressChanged              +=              new
ProgressChangedEventHandler(
        delegate(object o, ProgressChangedEventArgs args)
        {
            string b = args.UserState as string;

            lblProgress.Text = b;
```

```
        });

bw.RunWorkerAsync();

        }
    }
}
```

The Database

The index data is stored in an offline database that accompanies the program. It contains the information parsed from the master index files from SEC.gov.

The database is in the SQLite format. This database format was chosen because it is designed to be a lightweight offline system to accompany programs. It is often used in mobile applications to store resources needed by the program. Initially we used Microsoft SQL Server and Oracle MySQL in the project. However, we found that these database programs did not meet our requirements, since these programs need to have a running database server instance to make requests. This was incompatible with our desire to keep the entire program self-contained on a hard drive.

SQLite is functionally quite similar to SQL Server or MySQL. The main differences are certain differences in SQL code syntax, and a lack of a set and indexed date format. There are also substantial differences in search performance, especially in text matching. This is one of the reasons some procedures in the program have a long execution time.

The database consists one table named FormIndex containing the data on all forms available through Edgar. These items are:

- CompanyName: The full legal name of the company at time of form submission.
- CIK: The Central Index Key of the submitting company. A primary key that is unique to each business entity.
- Formtype: The type of the submitted form in its alphanumerical short form (i.e. 10-K 8-K 424B2 etc.). For an explanation of each form type, see sec.gov
- DateFiled: The day the form was registered as submitted.
- Filename: The location of the file on the SEC servers. Also contains the SEC accession number.

The database also contains a view that outputs all available years.

A view is a preconfigured query that is loaded each time the view is called. The view has the following query:

```
select distinct strftime('%Y',datefiled) as 'year' from Raportindex Order by strftime('%Y',datefiled)
```

It uses the “distinct” clause that makes each unique hit repeat only once. This reduces a list of the year of each form submitted, to a list of the years in the database. This view is used by several procedures to perform operations year by year. The lack of a dedicated date format requires using the strftime function to parse the date from a text format to a date-logic enabled format.

At time of writing (September 2014), the database contains the information on 14 092 692 records and occupies about 1,5GB of space. The database is unsecured, and can be accessed and edited by anyone that can gain access to the file. For database management, we have been using the SQLite Manager Plugin for the Mozilla Firefox web browser. A complete list of SQLite management software can be found on the SQLite website http://www.sqlite.org/cvstrac/wiki?p=ManagementTools . The database file is found at: <DriveLetter>:\EDGAR\EDGAR_Index.sqlite

The data in the database can be exported for use in a different program. Simply select the RaportIndex form in SQLite Manager, and press the “export” button to save the

data in CSV-format. This format can be opened in most data manipulation software such as Microsoft Excel, STATA and Minitab. Note that the sheer number of forms in the database may preclude the file from being opened in some programs.

The data can also be manipulated directly in SQLite Manager. This requires knowledge of SQL and the SQLite specific syntaxes used. A summary of the specifics of SQLite can be found at http://www.sqlite.org/lang.html, but using this site will require a basic understanding of SQL.

We have made the decision not to save the content of the forms into the database, but keep them as separate text files on the hard drive. This was done for two main reasons:

- Maintaining file integrity: The files are all stored as individual .txt files on the SEC server, and we wish to alter the forms as little as possible while downloading them. Our concern is that formatting and structure that could be useful in a search could be lost if the text is parsed into a database.
- Technical limitations: The largest file we have downloaded is more than 400MB large. This would have to be parsed into a single VARCHAR(MAX) cell in the database. The theoretical max size that can be fitted into a SQLite VARCHAR(MAX) cell is one billion characters, or bytes. 400MB is equal to 419 430 400 bytes, or a bit less than half the theoretical max size. We believe that file sizes will increase as more multimedia items are embedded into forms, and that the theoretical max size of will soon be reached. The size of the largest annual report increased from 250MB in 2012 to more than 400MB in 2013.

Many users might be more comfortable with individual files, than database items. There is also a risk that inexperienced users extract too many records at a time, and thereby risks crashing their computers. The risk of inadvertently opening a large number of text files by inexperienced users seems much smaller.

The most important argument against storing the forms as individual files is a possible decrease in search performance, and increased difficulty in determining the integrity of the forms. The form downloader has therefore been structured to do a full integrity check of the historical files each time it is run.

An Alternative Method of Structuring the Data

The program searches forms by opening all forms of the relevant type, and reading their entire content looking for the search term every time a search is made. This is a "brute force" way of searching that is quite slow and requires a lot computing power. Searches can take several hours potentially excluding time-sensitive users.

The alternative would have been to create an index of all words in all forms. This is done by creating a reference to every single word in every single document in a database. The database will comprise of a table that contains all the words ever used in any form, a table of all forms submitted (similar or identical to the one we have created for the program) and a hit table showing where a word is encountered in a form. Since there is a large degree of reuse of words across all forms, the list of unique words should be a lot quicker to query than all forms. When the words that are search for is located in the words table, a lookup can be made in the word-hit table.

Running searches in this way is a lot faster since the logical position of a hit in the dataset can be deduced from the primary key of the words in the search. This structure is similar to what Google and WRDS uses, and offers quick searches, and better possibilities to offer the system to a wider audience through a web portal.

The downside to this structure is that the indexing all forms is an immensely time, storage and computational power intensive procedure. It requires a centralized database, a web service and the associated user interface, and frequent administrator attention. Any administrators would need to be IT professionals. The code that parses the text would need to be a lot more sophisticated than it is to properly separate words, while ignoring HTM, and other code.

Overall, the costs of implementing such a structure are bigger than the benefits, for the purposes of this paper.

Mac Version

There have been made requests for a Mac OS X version of the program. This has not been a priority during this project, but the use of C# for .NET as coding language was partly made to facilitate porting to other platforms. By using the Mono framework, we believe that the program can be ported to Mac OS X and Linux. The Mono project compatibility tool indicates that the opening splash screen and the method to prevent sleep mode are the only unsupported methods in the project. The database driver (SQLite) is also listed as unsupported, but it should be supported through separate resource packs. Some procedures such as the file addresses will need to be recoded to match the Mac file address format as well. One will also naturally need a computer running Mac OS X to perform debugging.

Threading

Operations in a program are executed in the order indicated by the code. The program will read one line of code, do the operations that line command, and go to the next line. Usually, this is not a problem as most simple operations are executed so fast as to not be perceivable by humans, so that the queue of commands is executed before the next user input happens.

In the program, several methods take a long time to execute. In order to keep the user interface responsive, it is necessary to run these methods on their own “threads”. A thread is a separate execution path that executes the code it is ordered to do, while keeping the main program thread free to execute other code, such as handling user input and updating the user interface. The thread can send information to the main thread such as progress and completion status. This is evident in the program for example when during a search, the progress bar updates. A search without a separate thread would make the entire program appeared to have “hanged” or “crashed” until

the search completes, since any new commands would be in the back of the execution queue, behind the search.

The threading procedure can also be used to increase the performance by dividing execution paths on many simultaneous threads. When searching, the performance is to a large degree limited by the speed the hard drive can locate and load the file to memory. In a single thread search, the hard drive will remain idle while the program searches the file, and will only activate when the search is finished, and a command to load a new file is received. It also means that the search can also only use one processor core at a time.

The program uses a procedure that can split the search of individual files between a dynamically changing number of threads, based on what gives the highest performance. This method is embedded in newer versions of Visual Studio (.NET 4.0+), but as the program is in .NET 3.5, a user created method by Rob Volk is used with small modifications. This ensures that because the different thread will be in different states of execution, there will nearly always be a queue of requests to the hard drive for form files. If file-reading performance was not the limiting factor, a suitable number of threads to completely occupy the next performance bottleneck will be executed. This ensures that the maximum hardware limited search speed is reached, no matter the hardware configuration. It also spreads the search over all available physical CPU cores, where a single thread search can only use one core at a time. It might make the computer unavailable to execute other work while searching due to the large amount of computing resources occupied by the program.
Apart from a change in how the method is declared, this code is entirely the work of Rob Volk. All credit and big thanks goes to him.

```
public static void EachParallel<T>(this IEnumerable<T> list,
Action<T> action, BackgroundWorker bw)
        {     //Method retrived from http://robvolk.com/parallel-
foreach-loop-in-c-3-5/
            //Code by Rob Volk June 19. 2009
```

```
            // enumerate the list so it can't change during execution
            // TODO: why is this happening?
            list = list.ToArray();
            var count = list.Count();

            if (count == 0)
            {
                return;
            }
            else if (count == 1)
            {
                // if there's only one element, just execute it
                action(list.First());
            }
            else
            {
                // Launch each method in it's own thread
                const int MaxHandles = 64;
                for (var offset = 0; offset <= count / MaxHandles;
offset++)
                {
                    // break up the list into 64-item chunks because
of a limitiation in WaitHandle
                    var     chunk     =     list.Skip(offset     *
MaxHandles).Take(MaxHandles);
                    if (bw.CancellationPending == true)
                        return;
                    // Initialize the reset events to keep track of
completed threads
                    var          resetEvents          =          new
ManualResetEvent[chunk.Count()];

                    // spawn a thread for each item in the chunk
                    int i = 0;
                    foreach (var item in chunk)
                    {
                        resetEvents[i]             =             new
ManualResetEvent(false);
```

```
                        ThreadPool.QueueUserWorkItem(new
WaitCallback((object data) =>
                        {
                            int             methodIndex             =
(int)((object[])data)[0];

                            // Execute the method and pass in the
enumerated item
                            action((T)((object[])data)[1]);

                            // Tell the calling thread that we're
done
                            resetEvents[methodIndex].Set();
                        }), new object[] { i, item });
                        i++;
                    }

                    // Wait for all threads to execute
                    WaitHandle.WaitAll(resetEvents);
                }
            }
        }
```

Additional Helper Procedures

These code snippets are not integral to the main procedures of the program, but are necessary since both main procedures and the user interface use them. Many of these procedures could have been integrated into other methods, but are separate methods to enable re-using of the same code.

```
//                Method                retrived                from
http://www.codeproject.com/Questions/191236/Function-to-find-
Current-Quarter-of-the-Year

        public static int GetQuarter(this DateTime dt)
        {
            return (dt.Month - 1) / 3 + 1;
        }
        //Method                    retrived                    from
http://stackoverflow.com/questions/1288718/how-to-delete-all-files-
and-folders-in-a-directory
```

```
        // Code by Adam Robinson
        public static void Empty(this System.IO.DirectoryInfo
directory)
        {
            foreach (System.IO.FileInfo file in
directory.GetFiles()) file.Delete();
            foreach (System.IO.DirectoryInfo subDirectory in
directory.GetDirectories()) subDirectory.Delete(true);
        }
        public static List<string> GetFormTypes(string AppDir)
        {
            //Method that gets all the different types of SEC forms
contained in the database for populating the user interface
            SQLiteConnection conn = new
SQLiteConnection(GetConnectionString(AppDir));
            List<String> FormTypes = new List<string>();
            conn.Open();
            var cmd = conn.CreateCommand();
            cmd.CommandText = " Select Distinct Formtype from
raportindex order by formtype";
            var r = cmd.ExecuteReader();
            while (r.Read())
            {

                FormTypes.Add("" + r[0]);
            }
            return FormTypes;
        }

        public static string GetConnectionString(string AppDir)
        {
            //Method for centrally storing the connection string to
the database, so that if it needs to be altered
            //it only needs to be done once here.
            string cs = "Data Source=" + AppDir +
":\\EDGAR\\edgar_INDEX.sqlite;Version=3;";
            return cs;
        }
        public static string  Appdir()
        {
```

```
            //Method to find the drive letter of the hard drive the
program is run from
        string                          aplpath                          =
Path.GetDirectoryName(Application.ExecutablePath);
        string AppDirectory = aplpath.Substring(0, 1);
        return AppDirectory;
        }

        public static DateTime getmaxdate()
        {
            //Method that gets the newest date that is registred in
the database.
            DateTime Maxdate = DateTime.MaxValue;
            string AppDir = Appdir();
            string    cs    =    "Data    Source="    +    AppDir    +
":\\EDGAR\\edgar_INDEX.sqlite;Version=3;";
            SQLiteConnection conn = new SQLiteConnection(cs);
            var cmd = new SQLiteCommand();
            cmd.Connection = conn;
            conn.Open();
            cmd.CommandText    =    "Select    Max(Datefiled)    from
raportindex";
            var res = cmd.ExecuteReader();
            while (res.Read())
                Maxdate = Convert.ToDateTime(res[0]);
            return Maxdate;
        }
```

Dictionary on IT-Terms

The following section contains a quick explanation of several IT-terms that are used in the paper or code comments

Method: A block of code that is designed to achieve a purpose. Can achieve simple or complex tasks. One example can be the code that downloads all the forms from the database.

Parse: Converting data into a computer readable format. One example can be to convert a date from text into a data object where one can apply logic like adding a month.

Metadata: Data about data. In our thesis, it mostly applies to the index data about forms. Since a SEC-form is data, information such as address, date of submission, submitting company etc. is effectively data describing data.

Function: A set of code that takes an argument, does an operation on it, and returns a result. An example is the code that takes a date and returns the quarter of that date as a number.

Variable: An object that stores a piece of data in a certain format.

Int or Integer: The most common format for storing whole numbers in programming. Other number formats that might be relevant are Long and Decimal

String: The most common format for storing text in a program. Note that anything can be stored in a string, including numbers. Numbers stored in strings cannot have math logic applied to them.

Bool or Boolean: A variable that can only have two states, false or true. Is often used in conjunction with If-statements (see below).

If: Will perform an operation based on the condition of a statement. For example, the downloader uses an if-statement to control actions based on whether the file is already downloaded. If it is present (FileExists=TRUE) it will go to the next form. If it is not present (FileExists=FALSE) it will download the form to disk.

ForEach or For Each: Repeats an operation for each item in a list.

Hard coding: Giving a variable a set value in the program, rather than making it changeable by a user through the user interface. For example, the connection information to the database is written in the code, rather than being made an option.

User Interface: The part of the program the user sees and interacts with. By manipulating items such as buttons in the interface, the user can initiate, and change the execution of the underlying code. The underlying code can then report to the user interface through text boxes and progress bars. Is not needed for a program to function, but without it, all program variables will need to be hard-coded

SQL: Structured Query Language. A programming language used to perform operations in most database engines. Should not be confused with SQL Server, SQLite and MySQL, which are all database engines (programs).

HTM or HTML: The standard markup language used to make web pages. Is used in many SEC forms to add formatting options over plain text files.

Candidate Key: One or more variables that can be used to uniquely identify a record in a database. If the candidate key is used in the database to uniquely identify a record, it is a primary key.

References

Academic Textbooks

Bodie, Z., Kane, A., Marcus, A. (2011) Investments and Portfolio Management, global edition McGraw Hill, New York.

Fabozzi, F., (2012) The Handbook of Fixed Income Securities, 8th Edition. McGraw Hill, New York.

Feuerstein, S., (2007) Oracle PL/SQL Best Practices, 2nd Edition. O'Reilly Media, Sebastopol, California.

Pindyck R. & Rubinfeld, D. (2005) Microeconomics, International edition, Pearson, New Jersey.

Research Papers

Bienz C., Faure-Grimaud, A. and Fluck, Z. (2013) The Defeasance of Control Rights. Working Paper. Working paper.

Bradley, M. and Roberts, M. (2004) The Structure and Pricing of Corporate Debt Covenants. Working paper.

Engelberg, J., Sankaraguruswam, S.y (2007) How to Gather Data Using a Web Crawler: An application using SAS to search EDGAR. Working paper.

Julio, B. (2013) Corporate Investment and the Option to Repurchase Debt. Working paper.

Kahan, M., E. Rock (2009) Hedge Fund Activism in the Enforcement of Bondholder Rights, Northwestern University Law Review.

Legg, M.P, Tang, H. (2011) Why Casinos Are Not Recession Proof. Working paper.

Myers, S. (1977) "Determinants of Corporate Borrowing". Journal of Financial Economics.

Smith, C. Jr., Warner, J. (1979) On Financial Contracting: An Analysis of Bond Covenants. Journal of Financial Economics.

Sufi, A., Roberts, M. (2009) Renegotiation of financial contracts: Evidence from private credit agreements, Journal of Financial Economics.

Internet

Bloomberg L.C. (2014) Available at:

http://www.bloomberg.com/professional/solutions/education/ [15.10.2014]

Bowie, L,. “Cosmetic Center to merge Revlon subsidiary will be absorbed” (1996)

Baltimore Sun. Available at: http://articles.baltimoresun.com/1996-11-

28/business/1996333049_1_cosmetic-revlon-prestige [11.11.2014]

Fixed Income, Mergent, (2014) Available from:

http://www.mergent.com/mergent-solutions/fixed-income-data [01.11.2014].

Lambert, E., Doghouse On Wheels, Forbes, (2005) Available from:

http://www.forbes.com/free_forbes/2005/0131/094.html [15.10.2014]

Revlon Investor Relations, (2014) Available at: http://phx.corporate-

ir.net/phoenix.zhtml?c=81595&p=irol-irhome [01.12.2014]

Revlon Annual report 2004, (2005) Available at: http://library.corporate-

ir.net/library/81/815/81595/items/149265/RevlonInc2004Annual%20Report.pdf

[01.12.2014]

Security and Exchange commission, (2014) Available at:

https://www.sec.gov/investor/pubs/edgarguide.htm#.VH9JURGzlWU [15.10.2014]

Hudbay Minerals Inc. Consolidated Financial Statements Available at: http://www.hudbayminerals.com/Theme/HudBay/files/doc_financials/HudBay_yearendFinstats_4mar2009.pdf [01.11.2014]

Price Communications, 8-K Filing (2000) Available at: http://www.sec.gov/Archives/edgar/data/355787/000091205700052069/0000912057-00-052069.txt [03.12.2014]

Las Vegas Sands, 10-Q Filing (2002) Available at: http://www.sec.gov/Archives/edgar/data/850994/000085099403000001/lvsi_form10k-dec2002.htm [05.12.2014]

Lectures

Bienz, C. (2014) Lecture 5, FIE401 Empirical Finance, Bergen [03.10.2014]