Memory Error Prevention Through Static Analysis and Type Systems

University of Oulu
Faculty of Information Technology and Electrical Engineering / Degree Programme in Information Processing Science
Bachelor's Thesis
Henri Hyyryläinen
22nd May 2019

Abstract

Modern software is everywhere and much relies on it so it is important that it is secure and reliable. As humans, software developers make mistakes that may be really difficult to detect. In memory unsafe languages a large part of these mistakes are related to memory usage and management. In order to reduce the amount of bugs in software this thesis looks into using static analysis tools and other methods to automatically find where these mistakes are or alternatively preventing them altogether. This is done through a literature review. Unfortunately, static analysis results in many false positives that can take a long time for developers to sift through. For this reason many static analysis tools augment their usefulness by inserting dynamic, runtime checks in places where they are uncertain whether there is an error or not. One final approach, discussed in this thesis, for securing software memory usage, is to employ type systems or memory safe languages like Java that are designed so that the programmer is not allowed to access raw memory and make mistakes related to it. The large amount of checks that these kinds of languages must always do, result in a reduction in performance. As such all of these approaches have benefits and limitations regarding their use. The major findings were that much research has been done in static analysis tools that have managed to detect real problems. Many of the developed tools are unfortunately not available, and the ones that are available haven't been updated in a long time or they require complicated setup reducing their usefulness.

Keywords
static analysis, program analysis, memory error, type systems

Supervisor
Ph.D., University Lecturer Antti Siirtola

Contents

1. Introduction

Software is necessary for modern information infrastructure and as such it is natural that we require reliable software (Hiser, Coleman, Co, & Davidson, 2009). Zhogolev defines the reliability of software as being the ability to perform certain functions in specific conditions with a large enough percentage of success. With this definition a system that works most of the time under some conditions can be considered reliable. (as cited in Frolov, 2004.) Another very important quality attribute of software is security (Frolov, 2004). Software security is about preventing compromises to the integrity of software. A program in which the authentication functionality can be exploited is an example of a program with bad security. (Techopedia, n.d.). Safety is also an important property, it is defined as the software not performing any incorrect actions. These are some of the main reasons why focus should also be given to the tools and methods software developers use in order for them to produce better software. According to Hiser et al. (2009) current software development methods result in many different kinds of memory related errors. These errors can lead to very serious problems such as the loss of critical data or compromise of systems by an attacker, in addition to the more harmless program crashes.

The topic of this thesis is about the tools and language design features, that help in making reliable and secure software. This is important because tooling helps greatly in improving the reliability and efficiency of developed software (Dhurjati, Kowshik, Adve, & Lattner, 2005). This is especially looked at from the point of view of programming languages, like C++, that give a lot of freedom to the programmer to do whatever. Bugs happen when developing software and in order to prepare for this inevitability it is a good idea to employ static analysers (Black, 2012). Static analysers can detect many problems in software like performance bottlenecks, safety violations and security vulnerabilities, while it is still being implemented (Bodden, 2018). Black (2012) even goes as far as to argue that any ethical software development needs to make use of static analysis tools. For these reasons it is important to explore the available techniques and tools that software engineers can use to catch errors in the software they develop.

The rest of this introduction is dedicated to describing the major impact on security that memory related errors cause. The research methodology is discussed in the next chapter. The rest of this thesis is then split into the chapters discussing the different approaches and existing tools, then in the discussion chapter the major findings are summarised and discussed again. Then finally in the conclusion, the major points of this thesis are covered again and a possible direction for future research is presented, the limitations of this thesis are also discussed.

1.1 Impact on security

The aim of software security is that software continues to function correctly even while under attack (McGraw, 2004). Memory errors don’t always expose a way for an attacker to exploit them but even in these cases an attacker could potentially exploit these problems

in order to perform a very effective denial of service attack (Sun et al., 2018; Yong & Horwitz, 2003). However, when memory errors in programs exposed to the internet are exploitable, they can lead to the attackers gaining control over an entire system (Oiwa, 2009). So from a safety viewpoint it is also very important to make sure these types of errors are found and fixed. Or mitigated some other way, for example with dynamic checks around memory access.

C and languages similar to it employ manual memory management (Xu & Zhang, 2008). These types of languages are not memory safe languages as they require the programmer to explicitly request and free memory when needed instead of the language taking care of it automatically. C and C++ are examples of these kind of, memory unsafe languages (Kroes, Koning, van der Kouwe, Bos, & Giuffrida, 2018). Most common flaws in programs are memory related (Hiser et al., 2009). This does not apply to languages designed to be memory safe as their language design is made to counteract these issues. The most common issues, in not memory safe languages, are array bounds checking problems (buffer overflow and underflow) (Chess, 2002; Hiser et al., 2009; Kroes et al., 2018; Oiwa, 2009), not releasing memory (memory leak) (Heine & Lam, 2003; Sun et al., 2018; Xie & Aiken, 2005; Xu & Zhang, 2008), releasing memory too soon (also called dangling pointers) (Heine & Lam, 2003; Hiser et al., 2009; Oiwa, 2009), and problems with using uninitialized data (Hiser et al., 2009). Chess (2002) also describes race conditions as being very common. These can be classified as errors related to memory use, but not even memory safe languages guard against cases like this. In Java, for example, the programmer needs to explicitly ask for thread safety for many operations with the “synchronized”-construct.

Even though a lot of research has gone into fixing these types of issues they are still prevalent. Buffer overflows are particularly troublesome as they are quite often easy to exploit and use as a point of entry for a worm. (Kroes et al., 2018.) Memory leaks are another kind of issue that is still widely problematic in many widely-used programs (Xie & Aiken, 2005). A common attack is to exploit a buffer overflow and replace the return address to be in data provided by an attacker. This then causes the program to continue executing the attacker’s code. There are ways to protect programs against this type of attack, some of them have high runtime costs. (Yong and Horwitz, 2003.) In fact most attacks aren’t done by finding novel ways to compromise systems rather most attacks are repeated exploits of already known problems (Evans & Larochelle, 2002). Out of the 190 security attacks referred to by Evans and Larochelle (2002) only 4 are problems in cryptography, most of the rest are problems like buffer overflows and string format errors. In another study referenced by Ganapathy, Jha, Chandler, Melski, and Vitek (2003) buffer overruns where the top vulnerability in UNIX systems.

Static analysis and dynamic protection tools help in creating software that is resistant to attacks (Weber, Shah, & Ren, 2001), as a single mistake by a careless programmer, is able to undermine the security of the entire system (Ganapathy et al., 2003). These mistakes can be hard to catch without any help from tools. In addition to tools, secure software development can also be helped by an execution environment, for example the Java virtual machine. This isn’t without problems either as the Java virtual machine can also contain security vulnerabilities. (Pomorova and Ivanchyshyn, 2013.)

2. Methods

Turner (2018) was used as the basis for literature review methology. Initial searching for relevant articles was done with Oula Finna[1] and Google Scholar[2] using general terms from the title of this work: “static analysis memory error”. After identifying only a few relevant articles in the top twenty results, the search was continued on Scopus. The initial used search terms were “static analysis” and “memory error”.

Further searching on Scopus was done with four searches. The first paired static analysis with security or safety, programming and C++, the search was further limited to the computer science subject area. This resulted in 158 documents which were sorted by relevance. The next search was for static analysis combined with memory and safety and with C++ and programming. This resulted in 29 documents. The third search was static analysis paired with memory error. This search returned 21 documents. The fourth search was type system paired with memory and leak. This resulted in 23 documents

All of these results were examined based on the titles of the articles and ones that seemed relevant to any of researched topics, static analysis, type systems, memory safety in languages, were selected for further analysis. The articles citing Heine and Lam (2003) in Scopus were also examined similarly to other found results. There were 81 documents that had cited it.

Searching was also conducted on ACM for articles related to ACM search for “static analysis” memory error because “’static analysis’ AND ’memory error”’ only gave 8 results. This search had some good resources when sorted by relevance even though there were 52 169 results. Also on ACM, the search term “memory leak type system” was used, but it resulted in 364 146 results. When these were sorted by relevance the top 20 results contained several promising articles, which were further studied.

Some of these found, relevant articles were not readable immediately from Scopus through the SFX link. Some of these articles were found on ACM Digital Library. This search also resulted in more relevant articles found in the top 5 relevant results when searching for those articles. Relevant other articles were found when searching for the full text of the following articles: Di and Sui (2016), Rafkind, Wick, Regehr, and Flatt (2009), Ravitch and Liblit (2013), Lim, Park, and Han (2012), Zhang and Li (2005) (no article found but one relevant article was in the top results), Dhurjati, Kowshik, Adve, and Lattner (2003), Gjomemo et al. (2016) (no article found, other relevant results in top results), Kowshik, Dhurjati, and Adve (2002), Hutchins, Ballman, and Sutherland (2014) (no article found, other relevant results in top results).

Searching for information specific to CppCheck was done on Scopus with the search term “cppcheck” which returned 7 document results. From these results articles were selected for further study based on their abstracts, unfortunately the full text of one conference paper could not be located.

[1] https://oula.finna.fi/

[2] https://scholar.google.fi/

3. Static analysis

In this chapter the properties of static analysis are explored in relation to helping produce better software. Additionally static analysis is compared with dynamic analysis in terms of their strengths and weaknesses, it is also discussed how these approaches have been combined in the literature.

3.1 Comparison with dynamic analysis

It has long been believed that detecting errors in software before it is ran is beneficial (Das, 2006; Sokolov, 2007). Especially in safety-critical systems where detecting problems before runtime is absolutely vital (Dhurjati et al., 2005; Kowshik et al., 2002). Traditionally testing was the method for detecting errors in software but new tools such as static analysers and dynamic protection have been developed (Frolov, 2004). Both of these approaches have upsides and downsides. (Frolov, 2004; Weber et al., 2001.) Static analysis tools attempt to build a flow graph of all possible executions of a program (Grech, Fourtounis, Francalanza, & Smaragdakis, 2018; Sokolov, 2007). And from this analysis the tool attempts to detect problems. Whereas dynamic protection inserts guards into the program, that check, for example, the validity of pointers during runtime. The dynamic protection is also called dynamic analysis as it can be used to complement static analysis like it was used in Frolov (2004).

The advantage of static analysis over dynamic analysis is that it is complete. The completeness here means that the generated flow graph contains all possible executions of the target program. This is very effective but suffers from two major downsides: it is expensive to compute for substantial software, and it lacks precision. Precision is the ratio of incorrectly detected issues to the number of correctly detected issues. The lack of precision in static analysis is caused by the predicted flows not entirely matching actual flows when running the program. (Grech et al., 2018.) Whereas in dynamic analysis the observed program behaviour depends on the user input it is given (Sokolov, 2007) and thus is limited by how comprehensively the software is tested with different inputs.

The major downside of static analysis is the false positives, among valid reports, it usually generates. This causes wasted time spent investigating these false positives. Often when increasing the amount of actual problems static analysis tool finds, it also finds more false positives, which means that its usefulness doesn't necessarily increase. (Ciriello, Carrozza, and Rosati, 2013.) With many false positives developers might not feel like using static analysis is worth their time (Hovemeyer, Spacco, & Pugh, 2005). However many existing static analysis systems are lacking in branch awareness and thus their detection accuracy suffers and they might, for example, not be able to find memory leaks (Sun et al., 2018). Not filtering out infeasible paths through the program is a common source of this inaccuracy (Hovemeyer et al., 2005). This is a place for a lot of potential improvement. But Ciriello et al. (2013) goes as far as to claim that a perfect static analysis tool can never exist and all problems can not be found due to the nature of computing.

Frolov (2004) presents an approach where both of these approaches, to improving software quality, can be merged to reduce the downsides of each. In their approach, the dynamic protection is enabled in places where static analysis could not determine some operation to be safe. This way the dynamic checking is much cheaper and there is a reduced number of false positives, compared with plain static analysis.

Dynamic analysis, in the form of automated or manual testing done by running the program, can detect some aspects of the program much easier than static analysis, as with static analysis there are scalability problems when trying to infer all possible execution flows (Grech et al., 2018). Grech et al. (2018) propose a hybrid approach to static analysis where the static information is augmented with dynamic runtime information in order to reduce false positives and increase the performance of the analysis. They thus conclude that in program analysis there are three competing quality properties: completeness, precision, and scalability. Their approach is designed to focus on precision and scalability at the cost of completeness as the dynamic information they add to the static analysis process can't capture all possible program execution paths. Compared with earlier tools that they mention, their approach replaces parts of the static analysis with dynamic facts in order to improve the scalability. This dynamic information is derived from heap snapshots. A huge limitation in their work, in applicability to this thesis, is that their tools only work with the Java Virtual Machine (JVM). However their approach to not modeling the heap fully in static analysis and instead only relying on information from actual heap snapshots, without modeling writing operations, may be applicable to C++.

The major disadvantage of a dynamic analysis system is the degraded performance during runtime. Another disadvantage is that most systems cannot correct the program behaviour and for example can only terminate the program to prevent more harm from being done. The advantage is the simplicity of application. Combining this approach with static analysis allows reducing the number of places that need protection. Static analysis systems classify parts of a program in three categories. The first is unsafe fragments which are guaranteed to contain errors. The second is safe fragments which the static analysis could prove to be safe. The final category is the potentially unsafe fragments which could not be unambiguously classified into the other categories. This class needs runtime protection to alleviate the potential errors in them. A better static analysis might be able to reduce the number of these program fragments. Usually there is such a large amount of the potentially unsafe fragments that manually checking them is not feasible and this is where dynamic protection is a good tool for filling the gap. (Frolov, 2004.)

3.2 Description of static analysis

Static analysis tools are programming error detection tools that are ran on the static structure of a program. They can detect many kinds of bugs like memory leaks and out of bounds memory access (Ciriello et al., 2013). This analysis is performed before program execution. It can be done on different representations of the program, either on the source code or a compiled form. Analysing the source code is the most often used form as it is the most informative representation of a program. (Frolov, 2004.). However many of the papers referred to in this thesis have gone the route of running on machine code or byte code. One of the reasons for this is that there are situations when memory errors need to be detected in software without access to the source code (Hiser et al., 2009). For these cases, the memory error detection system introduced by Hiser et al. (2009) works on

binary executables. Running the analysis on compiled programs also has the advantage that the tools can work on many compiled languages much easier than tools that work on source code, as the source code of different programming languages can differ greatly.

Static analysers are used for scanning through large amounts of code for problems that would be nearly impossible to find manually (Ciriello et al., 2013). Memory leaks are one such example, which doesn't always affect program functionality but makes the software use much more memory than it actually needs (Ciriello et al., 2013; Sun et al., 2018; Xu & Zhang, 2008). Of course it is still possible for memory leaks to cause software to fail that is long running and memory-intensive (Heine & Lam, 2003; Xie & Aiken, 2005; Xu & Zhang, 2008). This might also lead to the performance degradation of an entire system and for example failure to launch new processes and excessive swapping slowing down everything running on that system (Sun et al., 2018). Though, once you know that there is a memory leak, it is possible to use analysis tools like Valgrind to point the programmer to the spot where the non freed memory was allocated.

Static analysis is a valuable tool, as through automating many manual inspections, it allows a small team to cope with large programs without excessive costs (Wendel & Kleir, 1977). With traditional code review the accuracy and number of problems detected is good but the desire to reserve humans for more difficult tasks lead to the creation of static analysis tools. Good tools can quickly identity weak spots and possible bugs in code, this is something that isn't possible with humans checking the code. Basic tools can detect some issues like floating point comparisons, more advanced tools are needed to detect issues with memory. (Sokolov, 2007.)

Static analysis is also important in keeping bugs away when software is being modified. Especially with legacy software, because even if it has tests, it still might contain many undetected defects (Ciriello et al., 2013). That can then surface when the software is modified (Ciriello et al., 2013). Another reason is that programmers can make mistakes when doing maintenance or upgrades to existing software (Landwehr, Bull, McDermott, & Choi, 1994; Sokolov, 2007, p. 223). When software is being modified it should be reviewed as carefully as when it was originally written (Landwehr et al., 1994, p. 223), but often this is skimped on. The situation gets even worse if the system that is being modified has no regression tests (Sokolov, 2007). This is another good spot to use static analysis to fill gaps in tests. Tools are a major help here as they can be used to automatically and repeatedly check millions of lines of code (Black, 2012). Though, the recommended solution to working with legacy software is to first write tests in order to be able to verify that the changed system works the same (Sokolov, 2007).

Static analysis is especially beneficial when the used programming language has a type system with which the compiler can ensure program correctness. A limitation in static analysis tools is the limited information their defect reports give, as they do not describe the scenario needed to trigger the problem. This leads to programmers not fixing some number of the problem reports (Das, 2006.). Also to be practical static analysis must fit seamlessly in the software development workflow (Ciriello et al., 2013).

3.3 Findings from the literature

The need for reliable software has resulted in a lot of research being done to identify ways to detect the violation of memory safety properties in programs. This property can be

generalized as requiring that each value created by event A must reach exactly one event B, in every program execution flow. This type of property is called source-sink property. With memory errors, the event A is the allocation of memory and event B is the respective deallocation of memory. In a program that fulfils this property, each memory allocation is freed exactly once. (Cherem, Princehouse, and Rugina, 2007.) This doesn't encompass every type of memory error, this is only for memory leaks and double frees, but extending this definition with additional access events that must happen between events A and B this model is also valid for other types of memory errors like use after free and the use of unallocated memory.

Weber et al. (2001) say that static analysis techniques, that existed at the time they wrote their paper, often worked on a really abstract level. For example by only just saying that a buffer overflow is possible, instead of being able to derive the program inputs that would result in that behaviour. Their technique also suffers from this limitation. But they claim that this is still useful information in the space of program security because, if it is possible to theoretically for the program to do a buffer overflow, it is good to fix it. Creating an accurate static analysis is not easy and in fact it is very easy for static analysis tools to miss problems when the analysed program contains complex branches (Sun et al., 2018).

Frolov (2004) uses a hybrid approach of combining static and dynamic analysis. In their approach, they use static analysis to prove some operations safe and use dynamic checks to protect other operations similar to Oiwa (2009). In addition they compare the advantages and drawbacks of static and dynamic analysis. With static analysis, the developers must address the issues, unlike with dynamic analysis that can insert automatic checks for preventing the errors, and this may be difficult as many issues found by static analysis can't be automatically decided. The program execution can depend on many environmental factors that the static analysis simply cannot predict. Thus it is necessary for the static analysis to cut corners here and not be entirely accurate in terms of all possible program executions. This can be mitigated with three approaches: asking the developer questions, finding a superset that also contains many false positives, and referencing a database with information regarding the runtime environment. All of these are labour intensive. Even the superset finding, as that results in a lot of false positives, that must be manually checked. The most promising of these is creating and maintaining the knowledge base about the program environment. In the knowledge base everything that the static analysis cannot infer, needs to be provided. For example, the results of system calls or network requests are the kind of information that needs to be added to the knowledge base. (Frolov, 2004.)

The final important finding of Frolov (2004) was that the hybrid approach they described can be iterated on. Once the approach has been used once the performance and reliability of the system can be further improved. This can be done by manually marking parts of the program safe in order to reduce dynamic checks. This helps the process get started as it can be very difficult to create a static analysis algorithm that can decide enough code fragments to be safe in order to not catastrophically affect the performance. Then it is possible to iteratively design the static analysis algorithm to classify more and more fragments unambiguously. In their example, they present the case of protecting against output format string vulnerabilities. First the analysis can be made to just check static variables in the current translation unit for the used format string. This will miss some safe uses, but will already limit the number of potentially unsafe fragments. To further improve this the analysis can be extended to include information from other translation units in order to be able to determine the format strings of more output function calls. (Frolov, 2004.)

Frolov (2004) also includes an experimental study that shows that even with just 4 steps and a code base of 500 000 lines of code the number of false warnings were reduced from 2618 to 35. In this experiment they only focused on format strings. Out of all the warnings only 0,15% were actual problems. At first they just counted all code fragments with output functions as unsafe, this resulted in 2618 potentially unsafe fragments. Then they implemented a basic analysis for detecting the use of constant format string in a single translation unit. This reduced the potentially unsafe fragments to 1150. Then they added detection of constant strings from other translation units and detection of print wrappers bringing the number of potentially unsafe fragments down to 35 that could then be manually examined in a day. They used a dynamic part to the analysis that protected the program from the start. It started at having a 7% impact on performance and 0% at the end when all of the calls through the protection layer were removed.

Static analysis can be used, in addition to catching memory errors, to verify that best practices and guidelines are followed in C++ (Sokolov, 2007). Sokolov (2007) also brings up the company culture aspects of static analysis tools. Mainly that they must be incorporated into the workflow in order to get used properly. It is no use having good tools if people only sporadically use them. Ciriello et al. (2013) presents similar models to Sokolov (2007). In their work, they explore the effects of applying the principle of continuous integration to static analysis. They call their approach continuous code static analysis. They describe it as running the static analysis, which can take multiple hours, in a similar fashion as continuous integration on a build server after code has been committed. In the case studied by Ciriello et al. (2013) they made the C++test tool run during the weekends and provided reports for the developers to fix for the next week's release, the company used a weekly release schedule. They also identified the need to keep false positives as low as possible. They also present that it is possible to track the direction the software is headed quality wise with continuous static analysis. This is done by tracking the number of issues detected from each software version. Then the direction can be determined from the last few points. If the trend is towards more issues then corrective measures can be implemented.

Bodden (2018) proposes that future static analysis tools could be developed to be self adaptive in order to improve their scalability. Scalability of current static analysis tools is a problem as the size of software grows. Current tools are written in general purpose programming languages with a limited set of customizability they can select based on the analysed program. To solve this problem Bodden (2018) proposes that static analyses should be created in a dedicated intermediate representation, for example as a graph problem. This provides much better chances for optimization than general programming languages. Additionally the analysis process should continuously adjust itself similarly to how just-in-time compilers work. The result would be that for each static analysis problem the static analysis tool would generate a highly optimized analysis. (Bodden, 2018.)

Bodden (2018) doesn't present a working implementation of their ideas. They only present the core idea in the hopes that the research community can help in the implementation of their ideas. They present the core concepts and the challenges they anticipate potential implementors will run into. At the core of the suggested algorithm is a new declarative definition language crafted specifically for creating static analysis tools. The design of this language needs to be balanced with regards to be able to express a variety of static analysis techniques but at the same time being restricted enough to offer potential for optimization. Then in their design this representation is compiled into a high level intermediate implementation that can be optimized. Then there would be also a low level representation that would be optimized in a way that takes the target program into account. Then this repre-

sentation is ran on the target program and profiling information is collected that can then be used to tweak the analysis to be more efficient, with help from a human. Then the final part of their design is the low level just-in-time compiler part that would constantly monitor the running analysis for bottlenecks and change to algorithms that are more efficient in the current situation. (Bodden, 2018.)

Lee, Hong, and Oh (2018) present a static analysis technique for detecting memory deallocation errors in C programs. In addition to detecting the issues their approach can automatically generate fixed code. This is based on solving an exact cover problem derived from their static analysis. The solution is a set of free statements that deallocate each piece of allocated memory exactly once. This makes it possible to make a fix that doesn't introduce new errors. This makes fixing these issues much easier as when manually fixing the programmer must consider all possible paths in order to not introduce a new problem, for example a double free, while trying to fix another problem.

Weber et al. (2001) is another paper where the authors developed a new tool that seems to have no source code available. They focused on developing a static analysis tool for detecting buffer overflows. They based their work on earlier analysis techniques that had serious limitations in terms of them not taking program flow or scoping into account. Weber et al. (2001) say that by taking program flow into account it reduces the number of false positives significantly. Their algorithm is based on determining the maximum lengths of data written to buffers and then determining if the buffer is big enough in all possible cases. A major limitation in their tool is that it must be given a list of potentially dangerous program statements to analyse, so it cannot be used on its own.

In general regarding static analysis, Weber et al. (2001) present that as program complexity increases the number of possible control flow paths increases dramatically. This is the reason why large programs cannot be accurately analysed with static analysis tools using a graph presentation, like used by Weber et al. (2001). In their experimental results, they show that their developed tool can reduce the number of hits from a tool that scans for potentially unsafe function calls by 25% to 60%. They only tested on three programs, resulting in the large variance of the results.

In contrast to the other papers discussed Hovemeyer et al. (2005) present a simpler analysis that tries to just find null pointer problems. They show that their tool is effective at finding actual problems without a huge number of false positives, despite the analysis being simple. Their tool is for analysing Java bytecode and it is still available. Their analysis focuses on reducing the amount of false positives by conservatively assuming that unknown values aren't errors. They also greatly limit the size of their program flow graph by leaving out complex interaction between functions and determining actually executed polymorphic functions. Instead their tool assumes that many variables are unknown and any concrete method in the case of polymorphic methods could be executed, but once again they limit the amount of warnings they give as their tool doesn't have enough information to give good warnings. They do mention that their tool will generate warnings from polymorphic code where all the possible implementation methods can cause a problem.

4. Existing tools for error detection

Many of the articles referenced in this thesis present static analysis or hybrid static and dynamic analysis tools that the authors developed. Sadly, it seems that none of these tools, that work on C++ code or x86 machine code, are available *and* open source. This is the most major limitation in the existing literature. Still, a few tools were found, but they are unmaintained and bordering on obsoleteness, or they were for a memory safe language. It seems that many of the papers set out to only create a proof of concept implementation of their algorithms and not to create a useful tool ensuring that others could benefit easily from their work. As such if anyone wants to use the findings of these papers to improve their software, they would have to reimplement the tools from scratch.

However, Lee et al. (2018) reference an earlier tool that has source code available, but the improved tool they themselves created is not available in source code form. Though, there are existing tools, they are proprietary. An example of this is that Ciriello et al. (2013) present an approach where a static analysis tool is automatically ran on committed code on a server. Then the generated reports from the long tool runs, over the weekend, are delivered to the developers. This is an example of how the tool can be incorporated into a workflow. Unfortunately, the tool, C++test by Parasoft, they used is proprietary. Ciriello et al. (2013) describe the tool as aiding developers prevent software defects by utilising rules that are tuned to find code patterns that lead to problems. As a proprietary tool, it's impossible to say what approaches are used by it and thus it is not possible to use their results in building a base for future research. In their tests, 14% of reported issues were false positives.

CppCheck is also an existing tool similar to C++test but it is open source and can be studied to find what types of problems it can detect and how. No available articles testing the effectiveness of CppCheck were found. Such articles, exploring its effectiveness, exist but I didn't have full text access to them.

The tool presented by Lee et al. (2018) is available, but only in binary form with no source code easily findable. Lee et al. (2018) mentions LeakFix tool that has a similar goal as their tool and that has source code available. It is even quite recent with the source code dating only back to 2015. However, the setup process requiring building their modified source version of an outdated compiler makes this less useful.

The compiler for memory safe C developed by Oiwa (2009) is available but the last update to the project has been in 2010 and it thus depends on outdated outside libraries. I was unable to build this program due to the Bohm garbage collector not being detected by the configure script. It is probably possible to update the code to work again, but it may take some considerate amount of work. Alternative outdated versions of the required libraries might allow building the software. Though even then the compiler is outdated as multiple new C standards have been published since its last update.

Sun et al. (2018) say that in their experience existing static analysis tools, that collect information from scanning source code, are lacking in accuracy and efficiency. Kroes et

al. (2018) agree that overheads caused by existing tools are unacceptable. They also say that the existing tools suffer from poor compatibility. This is a huge problem, as mentioned earlier, that low accuracy results in a lot of false positives and wastes time of developers.

Grech et al. (2018) mention HeapDL and Tamiflex tools, in addition to the tool developed in their paper, that implement some static checking for Java programs. These tools were not researched further as the focus of this thesis is to explore the feasibility of static analysis mainly for memory unsafe languages.

5. Type systems

In this chapter type systems are explored as an alternative to dynamic or static analysis in order to prevent errors in memory usage.

5.1 Type systems for memory safety

Another approach to preventing memory errors is from a language design perspective. Many languages can get by without dynamic memory allocation and some others have type systems built into the language for guaranteeing that dynamic memory is properly used, a recent example of this that has gained a lot of publicity recently is Rust. In addition, there are languages that have been designed to be memory safe. Examples of these kind of languages are Java (Oiwa, 2009) and C# (Hiser et al., 2009). In these kinds of languages runtime checks, for example bounds checking for arrays, and garbage collection are used to provide the memory safety (Dhurjati et al., 2003). This leads to significant overheads in performance compared with memory unsafe languages (Dhurjati et al., 2003). Because of this, many developers decide not to use these kinds of languages (Hiser et al., 2009). There are also legacy systems that would be ridiculously expensive to rewrite in a memory safe language (Hiser et al., 2009). For these reasons it is very interesting to find out if static analysis makes programs more secure without rewriting them in a memory safe language and taking a performance hit. As previously mentioned a hybrid approach can be used to lessen the performance impact of dynamic checks.

Even when a memory safe language is used it might be the case that the software needs to use some utility library that is written in an unsafe language. Java programs, for example, can use C libraries. This makes it possible that a misbehaving library causes major issues. (Ramasamy, Singh, and Singal, 2016.) Both of these types of languages thus have tradeoffs, but both are needed in at least some situations.

The C programming language has no memory safety, it does not prevent out-of-bounds memory access. Nevertheless such checks can be added to it. They just cost a lot of runtime performance. An improvement can be made to this by using static analysis to prove some memory accesses safe and remove checks around them. (Nazaré et al., 2014.) Sometimes adding this static component doesn't even require any new language syntax as Dhurjati et al. (2003) showed that a significant subclass of valid C programs can be proven statically to be completely memory safe. In other papers, existing languages have been modified in order to be safe.

5.2 Findings

Penna (2005) developed a new type system for C++ inspired by the safety of Java. It is very similar to the already mentioned static analysis techniques and dynamic protection as their approach is a combination of the two: it either statically proves memory access

safe or adds dynamic checks to memory access. However, the work is presented as an addition to the C++ type system making it possible to, for example, cause compilation errors from basic memory use related issues. Penna (2005) presents complex rules that pointers must follow in order to be valid, some of these rules need a runtime component if not enough information is available in the context of a memory access. They claim that this approach of only adding memory access checks when static checking fails offers superior performance over Java where each memory access is always checked.

Oiwa (2009) presents a memory safe C compiler that is compatible with the C language standard. Compared with usual C compilers that don't provide protection against out-of-bounds memory access their approach of combining static and dynamic checks guarantees total memory safety. They claim that their approach is the first one that allows standard C to be made memory safe. Penna (2005) developed a similar system for C++ that relies on statically proving some memory accesses and then falling back on runtime checks in some cases. Their goal was to give C++ the safety of Java without also ending up with as much runtime overhead. Yong and Horwitz (2003) also present a system that consists of a static analysis and a runtime checking component that guarantees memory safety.

The compiler created by Oiwa (2009) for C, is still available, and provides full memory safety for C code compiled with it. Unlike Penna (2005) their work also includes benchmarking details about the effectiveness of their implementation and thus give insight into how feasible this kind of approach is. Oiwa (2009) mentions that alternatives to their approach are runtime library features for buffer-overrun detection and creating new C-like languages that have memory safety included in the language design. One of the major benefits of an approach of this kind is that it doesn't require any modifications to standard compliant programs in order to use. However there are some classes of programs, like kernels, that are incompatible with the memory checking of this approach. (Oiwa, 2009.) Oiwa (2009) additionally used garbage collection in their implementation instead of letting the programmer deallocate memory. And their approach, unlike the approach by Penna (2005), also affected integer calculations as they needed a specific fat integer type that can hold the extra information that storing their fat pointers into an integer variable needs. They had to use this technique because ANSI C requires integer type to be able to hold pointers without losing information.

Oiwa (2009) measured that their implementation causes programs, that use pointers, to take around 2 to 4 times longer than when compiled with GCC. The performance overhead for integer math was 2%. From additional benchmarks, they determined that the way a program is written greatly affects the speed penalty imposed by the dynamic checks. In one benchmark where their solution was unable to prove allocating some structs on the stack was safe, they observed 6 times worse performance.

Heine and Lam (2003) present an addition to the C++ and C type systems that ensure there are no memory leaks and each object is deleted once. This system is based on a model of a single owner for each piece of memory where the owner is responsible for deleting or transferring ownership. This is similar to the memory model in the Rust language, see for example Klabnik and Nichols (2018). One additional restriction that Heine and Lam (2003) describe for this model, is that a field in an object that is a pointer must always or never own the memory it is pointing at, at public method boundaries. (Heine and Lam, 2003.) This means that whether a pointer is owning or not must be decided beforehand for each field in an object. This way there is no confusion stemming from the potential pitfall that a field could either be owning or not depending on the object's state. Heine

and Lam (2003) also developed a tool for detecting the violations of the introduced rules, in order to check the validity of their findings, in a large subset of C and C++ language. Unfortunately, this tool was not found to be available for use, which is likely explained by the fact that it was meant more of a proof of concept than an actual tool for widespread use. A major limitation of Heine and Lam (2003) is that their memory model does not prevent non-owning pointers from pointing to deleted memory.

Tlili, Yang, Ling, and Debbabi (2008, p. 377) took a different approach in that they extended the C language type system to include the necessary information for static checks. Of note, however, is the fact that they also have a dynamic component in their checking. So even with their changes to the language they couldn't come up with a way to remove runtime checks. Kowshik et al. (2002) present a variant of C language that is a subset of it, which has been designed so that memory accesses can be statically verified to be safe. Their approach does not need any runtime checks.

6. Discussion

Memory unsafe languages are still widely used. Some of the use cases for them are in software where the performance impact of a memory safe language design needs to be avoided (Dhurjati et al., 2003) or in legacy systems that would be very expensive to recreate (Hiser et al., 2009). As such memory unsafe languages like C and C++ (Kroes et al., 2018) should be made safer. The most common issues are memory related (Hiser et al., 2009). For that reason methods for making memory usage safe were explored in this thesis. Evans and Larochelle (2002) explored the statistics of security issues in software and they confirmed that memory related issues were at the top. Ganapathy et al. (2003) reported similar findings. Even though these articles are already pretty old, their findings are still relevant as similar high profile security issues are still being found. A somewhat recent example of a memory related issue is CVE-2014-0160 also known as Heartbleed ("CVE-2014-0160." 2013). It wasn't entirely similar to the issues covered in this thesis, but it was an issue with memory buffer reading. So even though much research has been done to try to reduce these types of issues they are still prevalent (Kroes et al., 2018).

Out of all the tools developed for the papers looked at in this thesis only a few were still available. Many of them were for memory safe languages or they didn't have source code available. The most promising of the tools was the compiler created by Oiwa (2009). Unfortunately, it is unmaintained and starting to show its age by not even compiling with up to date dependencies, the language version it was built against is also outdated. After all the articles were selected a potentially usable tool was referenced by Lee et al. (2018) however the article that described this tool was not found during the literature search and the tool was left out.

Even though no immediately usable tool implementations were found the reviewed articles presented a very comprehensive theoretical basis behind static analysis and the benefits different approaches have. However out of the introduced methods none were super promising in that most of them had issues with false positives and the rest had issues with false negatives. So further work could still be done on automating false positive detection even further. Additionally (Ciriello et al., 2013) and (Sokolov, 2007) brought up the workflow aspects of static analysis. If the tools, which don't exist currently, can't be integrated into the workflow their usage will be spotty and thus their benefits will be greatly reduced.

Heine and Lam (2003) references auto_ptr which is an outdated feature of the C++ standard library. The replacements for it are shared_ptr and unique_ptr types. In the case of unique_ptr, it implements the characteristics of the type system described by Heine and Lam (2003). From this it can be said that modern C++ development has embraced the lessons learned from Heine and Lam (2003) and has resulted in greater memory safety than in earlier versions of C++ as the use of smart pointers is recommended by the C++ Core Guidelines. It is possible to make static analysis tools for checking that best practices are followed (Sokolov, 2007) this would allow the safety of C++ code to be improved greatly if the core guidelines are automatically enforced.

The type systems looked at in this thesis were made not to require any syntax changes to C or C++. But that resulted in them not being able to be fully static without any runtime checks, for example see Oiwa (2009). In my opinion, a more interesting approach would have been similar to Rust. In Rust, you have to prove to the compiler that memory is properly managed in your program (Klabnik & Nichols, 2018). Which means that the Rust compiler can statically prove all correct Rust programs not to have the type of memory issues C and C++ commonly suffer from. However, the Rust compiler is also not fool-proof in the sense that complicated code may need to be written in an inefficient way or using unsafe language constructs.

7. Conclusion

In this chapter the major points of this thesis are summarised, limitations are briefly discussed and a direction for future research is presented.

7.1 What was learned

This thesis set out to find ways to utilize static analysis and type systems in order to reduce errors in software. The very common category of errors related to memory usage was the main focus, because they can cause very serious security issues and they reduce the stability of software.

The main findings where that it is possible to use static analysis to prevent memory errors. However, there were a lot of different approaches so it isn't possible to make a recommendation as to which one to use as a basis for future tools. Type systems and memory safe languages are also a way to prevent memory errors. Type systems can be used to limit the possible programs expressed in a programming language to make it impossible to make memory errors.

A lot of theoretical basis is covered in the earlier research. But there is a definite lacking of free to use, open source tools that developers could implement into their workflows in order to gain the benefits presented in the literature. Some tools were found but they were outdated and didn't offer an easy way to incorporate them to workflows.

7.2 Limitations

It is likely that not all articles about static analysis were found, so it is possible that there were important aspects that were missed. Definitely one article describing the LeakFix tool was missed, as such it wasn't analysed in this thesis. Additionally it is possible that some of the articles, that developed a static analysis tool, in fact do have their code available, it was just too well hidden and didn't show up in the top results on Google. Or there may exist other tools that were not covered by the found literature. It is also possible that the open source tool CppCheck may have impressive static analysis features, but the full text of any of the articles testing it was not found. A limitation regarding all the found articles is that, due to their large volume, only the most interesting or different findings have been discussed in this thesis.

7.3 Future research

The theoretical background is comprehensive, but concrete, usable tools are lacking. Without an effort to make an open source implementation the theories can't be put into practice without requiring everyone who wants to use them to reimplement them from scratch

each time, which is a colossal waste of time. There was also lack in comparative studies between the different approaches. Most of the articles also used their tools on different software, so it is difficult to compare the effectiveness of the different approaches.

This thesis is planned to serve as a basis for a follow up where design science research methodology will be used to implement a concrete, open source tool that could then be easily integrated into the workflows of software engineers. Literature is needed as a foundation for a larger study (Turner, 2018). As such literature review was a good methodology for this bachelor's thesis in order to serve as a basis for the planned follow-up.

8. Bibliography

Black, P. (2012). Static analyzers: Seat belts for your code. *IEEE Security & Privacy*, *10*(3), 48–52. doi:10.1109/MSP.2012.2

Bodden, E. (2018). Self-adaptive static analysis. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results* (pp. 45–48). ICSE-NIER '18. doi:10.1145/3183399.3183401

Cherem, S., Princehouse, L., & Rugina, R. (2007). Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 480–491). PLDI '07. doi:10.1145/1250734.1250789

Chess, B. V. (2002). Improving computer security using extended static checking. In *Proceedings 2002 IEEE Symposium on Security and Privacy* (pp. 160–173). doi:10.1109/SECPRI.2002.1004369

Ciriello, V., Carrozza, G., & Rosati, S. (2013). Practical experience and evaluation of continuous code static analysis with C++Test. In *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation* (pp. 19–22). JAMAICA 2013. doi:10.1145/2489280.2489290

CVE-2014-0160. (2013). Available from MITRE, CVE-ID CVE-2014-0160. Retrieved May 15, 2019, from http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160

Das, M. (2006). Unleashing the power of static analysis. In K. Yi (Ed.), *Static Analysis* (pp. 1–2). Berlin, Heidelberg: Springer Berlin Heidelberg.

Dhurjati, D., Kowshik, S., Adve, V., & Lattner, C. (2003). Memory safety without runtime checks or garbage collection. *SIGPLAN Notices*, *38*(7), 69–80. doi:10.1145/780731.780743

Dhurjati, D., Kowshik, S., Adve, V., & Lattner, C. (2005). Memory safety without garbage collection for embedded applications. *ACM Transactions on Embedded Computing Systems*, *4*(1), 73–111. doi:10.1145/1053271.1053275

Di, P., & Sui, Y. (2016). Accelerating dynamic data race detection using static thread interference analysis. In *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores* (pp. 30–39). PMAM'16. doi:10.1145/2883404.2883405

Evans, D., & Larochelle, D. (2002). Improving security using extensible lightweight static analysis. *IEEE Software*, *19*(1), 42–51. doi:10.1109/52.976940

Frolov, A. M. (2004). A hybrid approach to enhancing the reliability of software. *Programming and Computer Software*, *30*(1), 18–24. doi:10.1023/B:PACS.0000013437.87730.e5

Ganapathy, V., Jha, S., Chandler, D., Melski, D., & Vitek, D. (2003). Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (pp. 345–354). CCS '03. doi:10.1145/948109.948155

Gjomemo, R., Phung, P., Ballou, E., Namjoshi, K., Venkatakrishnan, V., & Zuck, L. (2016). Leveraging static analysis tools for improving usability of memory error sanitization compilers. In R. Bilof (Ed.), *Proceedings - 2016 IEEE International*

Conference on Software Quality, Reliability and Security (pp. 323–334). QRS 2016. doi:10.1109/QRS.2016.44

Grech, N., Fourtounis, G., Francalanza, A., & Smaragdakis, Y. (2018). Shooting from the heap: Ultra-scalable static analysis with heap snapshots. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 198–208). ISSTA 2018. doi:10.1145/3213846.3213860

Heine, D. L., & Lam, M. S. (2003). A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (pp. 168–181). PLDI '03. doi:10.1145/781131.781150

Hiser, J. D., Coleman, C. L., Co, M., & Davidson, J. W. (2009). MEDS: The memory error detection system. In F. Massacci, S. T. Redwine, & N. Zannone (Eds.), *Engineering Secure Software and Systems* (pp. 164–179). Berlin, Heidelberg: Springer Berlin Heidelberg.

Hovemeyer, D., Spacco, J., & Pugh, W. (2005). Evaluating and tuning a static analysis to find null pointer bugs. *SIGSOFT Software Engineering Notes*, *31*(1), 13–19. doi:10.1145/1108768.1108798

Hutchins, D., Ballman, A., & Sutherland, D. (2014). C/C++ thread safety analysis. In R. Bilof (Ed.), *Proceedings - 2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation* (pp. 41–46). SCAM 2014. doi:10.1109/SCAM.2014.34

Klabnik, S., & Nichols, C. (2018). *The Rust Programming Language*. Retrieved January 15, 2019, from https://doc.rust-lang.org/book

Kowshik, S., Dhurjati, D., & Adve, V. (2002). Ensuring code safety without runtime checks for real-time control systems. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (pp. 288–297). CASES '02. doi:10.1145/581630.581678

Kroes, T., Koning, K., van der Kouwe, E., Bos, H., & Giuffrida, C. (2018). Delta pointers: Buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference* (22:1–22:14). EuroSys '18. doi:10.1145/3190508.3190553

Landwehr, C. E., Bull, A. R., McDermott, J. P., & Choi, W. S. (1994). A taxonomy of computer program security flaws. *ACM Computing Survey*, *26*(3), 211–254. doi:10.1145/185403.185412

Lee, J., Hong, S., & Oh, H. (2018). MemFix: Static analysis-based repair of memory deallocation errors for C. In L. G. Garci A. Pasareanu C.S. (Ed.), *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 95–106). doi:10.1145/3236024.3236079

Lim, W., Park, S., & Han, H. (2012). Memory leak detection with context awareness. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium* (pp. 276–281). RACS '12. doi:10.1145/2401603.2401664

McGraw, G. (2004). Software security. *IEEE Security & Privacy*, *2*(2), 80–83. doi:10.1109/MSECP.2004.1281254

Nazaré, H., Maffra, I., Santos, W., Barbosa, L., Gonnord, L., & Quintão Pereira, F. M. (2014). Validation of memory accesses through symbolic analyses. *SIGPLAN Notices*, *49*(10), 791–809. doi:10.1145/2714064.2660205

Oiwa, Y. (2009). Implementation of the memory-safe full ANSI-C compiler. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language*

Design and Implementation (pp. 259–269). PLDI '09. doi:10.1145/1542476.1542505

Penna, G. D. (2005). A type system for static and dynamic checking of C++ pointers. *Computer Languages, Systems & Structures, 31*(2), 71–101. doi:https://doi.org/10.1016/j.cl.2004.05.002

Pomorova, O. V., & Ivanchyshyn, D. O. (2013). Assessment of the source code static analysis effectiveness for security requirements implementation into software developing process. In *2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS)* (Vol. 02, pp. 640–645). doi:10.1109/IDAACS.2013.6663003

Rafkind, J., Wick, A., Regehr, J., & Flatt, M. (2009). Precise garbage collection for C. In *Proceedings of the 2009 International Symposium on Memory Management* (pp. 39–48). ISMM '09. doi:10.1145/1542431.1542438

Ramasamy, S., Singh, A., & Singal, D. (2016). Enhancing the security of C/C++ programs using static analysis. *Indian Journal of Science and Technology, 9*, 1–4. doi:10.17485/ijst/2016/v9i44/104031

Ravitch, T., & Liblit, B. (2013). Analyzing memory ownership patterns in C libraries. In *Proceedings of the 2013 International Symposium on Memory Management* (pp. 97–108). ISMM '13. doi:10.1145/2491894.2464162

Sokolov, S. (2007). Bulletproofing C++ code. *Dr.Dobb's Journal, 32*(2), 37–42. Retrieved February 1, 2019, from https://search.proquest.com/docview/202693203?accountid=13031

Sun, X., Xu, S., Guo, C., Xu, J., Dong, N., Ji, X., & Zhang, S. (2018). A projection-based approach for memory leak detection. In L. O'Conner (Ed.), *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)* (Vol. 02, pp. 430–435). doi:10.1109/COMPSAC.2018.10271

Techopedia. (n.d.). Software Security. Retrieved May 15, 2019, from https://www.techopedia.com/definition/24866/software-security

Tlili, S., Yang, Z., Ling, H. Z., & Debbabi, M. (2008). A hybrid approach for safe memory management in C. In J. Meseguer & G. Roşu (Eds.), *Algebraic Methodology and Software Technology* (pp. 377–391). Berlin, Heidelberg: Springer Berlin Heidelberg.

Turner, J. R. (2018). Literature review. *Performance Improvement Quarterly, 31*(2), 113–117. doi:10.1002/piq.21275

Weber, M., Shah, V., & Ren, C. (2001). A case study in detecting software security vulnerabilities using constraint optimization. In A. D. Williams (Ed.), *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation* (pp. 1–11). doi:10.1109/SCAM.2001.972661

Wendel, I. K., & Kleir, R. L. (1977). FORTRAN error detection through static analysis. *SIGSOFT Softw. Eng. Notes, 2*(3), 22–28. doi:10.1145/1012319.1012323

Xie, Y., & Aiken, A. (2005). Context- and path-sensitive memory leak detection. *SIGSOFT Software Engineering Notes, 30*(5), 115–125. doi:10.1145/1095430.1081728

Xu, Z., & Zhang, J. (2008). Path and context sensitive inter-procedural memory leak detection. In *2008 The Eighth International Conference on Quality Software* (pp. 412–420). doi:10.1109/QSIC.2008.12

Yong, S. H., & Horwitz, S. (2003). Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on*

Foundations of Software Engineering (pp. 307–316). ESEC/FSE-11. doi:10.1145/940071.940113

Zhang, G., & Li, X. (2005). Static detection to dynamic memory errors. *Jisuanji Fuzhu Sheji Yu Tuxingxue Xuebao/Journal of Computer-Aided Design and Computer Graphics*, *17*(3), 400–406.