Project Description

Dr. David H. White & Prof. Dr. Gerald Lüttgen

Please note that some sections of the DFG project description template have been omitted here. This is because these sections are not applicable to our proposal.

1 State of the Art and Preliminary Work

Software programs making heavy use of pointers are notoriously difficult to analyse. To do so, one needs to understand which dynamic data structures and associated operations the program employs. Analysis tools for pointer programs, such as those based on shape analysis [65] and pointer graph abstraction [34], rely on an abstraction methodology that must be crafted for each specific data structure, and thus require *a priori* knowledge of the program to be analysed.

Knowing the shape of a data structure might, however, be insufficient for understanding its behaviour. For example, to recognise a linked list implementing a stack, the operations that manipulate the data structure are of key importance. Static analyses typically provide only approximations for this type of behaviour, due to imprecision in the analysis. The task is further complicated when dealing with legacy code, programs with unavailable source code and, even worse, programs with obfuscated semantics such as malware. Hence, the question we wish to address in this project is whether pointer programs can be understood with high confidence via a dynamic analysis that identifies dynamic data structures and the operations that manipulate them from an execution trace of the program under analysis.

In prior work, we have designed and implemented a proof-of-concept approach that achieves this goal for some limited scenarios [PSP1]. Our approach employs techniques from machine learning and pattern matching to identify the dynamic data structures and the operations that manipulate them. However, there are many limitations to our current approach that only allows the analysis of very simplistic software. Specifically, the analysis of nested data structures, or any data structure implemented using a recursive coding style, is not possible. These limitations preclude the analysis of most real-world software, e.g., the nested list structures commonly seen in device drivers or the recursive coding style used to implement many tree data structures. Furthermore, the analysis currently works on source code, thus excluding programs for which only object code is available or any setting where an analysis on object code is preferable.

We now seek to significantly improve and enhance our proof-of-concept approach to handle large, realistic classes of pointer programs by addressing the shortcomings outlined above, and show that the approach has application in many domains and can be applied to many different types of software. The present proposal was inspired in a large part by discussions with our international partners at the University of York, UK, in the context of the DFG-funded *initiating international collaboration* project on "Advanced Heap Analysis and Verification" (grant no. LU 1748/2-1).

Specific classes of software that we will target in this project are operating systems software, application software and legacy/obfuscated software, with (a) available source code and (b) available object code. We have already suggested that such an analysis will have application for automated verification, where it can be used to inform verification engineers and tools so that they can better select and configure their techniques [54, 61–63, 72]. Specifically, information on a program's data structures will enable the automatic verification of a significantly larger class of programs than is currently possible. When the source code is available, a high-level program understanding as gained by our approach will also be useful in optimisation [40, 59] and for the indication of programming errors [38] such as non-standard combinations of operations manipulating a data structure. In addition, the automated discovery and identification of data structures will be of great help to engineers maintaining legacy software.

However, it is when this type of analysis is applied to object code that its utility will increase even further. There will be significant challenges in performing a similar analysis on object code as much of the information about the program semantics, such as type information, is missing in an object code representation. It will be necessary to employ techniques from the area of object code analysis to

recover this information [4, 22, 43, 44, 68]. An analysis at object code level will be of great value for the analysis of kernel or device driver code. The natural end point is application to the reverse engineering domain, where effort may have been made to intentionally obscure a program's semantics. In addition, a robust reproducible understanding of the program at an abstract level enables the creation of a program signature, which can be of use in the domain of virus and malware detection [22].

In the sequel, we first describe the preliminary research we have performed to produce our proof-of-concept approach, and then follow this by a discussion of relevant work by others.

Our Preliminary Work. Identifying (or in machine-learning terminology: *labelling*) operations appearing in a program trace is a difficult problem. The initial obstacle is simply locating data structure operations, i.e., determining which *events* (e.g., a pointer write) in the trace (an *event trace*) correspond to an operation and which do not. This problem is compounded by the fact that invocations of the same operation may look very different: clearly, the addresses appearing in pointer variables will differ, but there may also be significant differences in the control path taken due to traversal or corner cases, such as inserting into an empty data structure.

The idea is to locate an operation by learning the repetition in the program trace caused by multiple invocations of that operation. For this to work, we must construct an abstraction of the trace that lessens the differences between invocations, and thus exposes the repetition. Moreover, we have made a couple of assumptions for our preliminary approach [PSP1] to be feasible, which are typically satisfied in real-world programs. Firstly, there should be a sufficiently large number of invocations to expose the repetition, and secondly, the surrounding context of the invocations should vary, otherwise the context could be included in the repeating pattern. We expect to be able to lift these assumptions to some degree as our machine learning and pattern matching components improve.

Our approach employs two abstractions; the first is the points-to graph, which represents the connectivity of program data in terms of pointers. By constructing a points-to graph describing the memory state after each event, we build the *points-to trace*. However, the points-to trace is unsuitable as input to the machine learning step for locating repetition, since the specific information about an event is captured very inefficiently. Therefore, we construct a second abstraction for each points-to graph, which captures the semantics of the event; using machine learning terminology we term this abstraction a *feature*. Thus, the search for repeating structure takes place on the *feature trace*, where the goal is to learn the set of *patterns* that best captures the repetition. The notion of "best" is determined by a *minimum description length* [32] criterion that evaluates how successful a set of patterns is at compressing the feature trace, and the search is performed using a genetic algorithm [31].

However, merely locating repetition in the program trace is insufficient as it is highly likely that repetition resulting from non-operations will also be discovered. Furthermore, because there are many different ways to code a data structure operation, it is unlikely that it will be possible to assign a label at the granularity of repeating pattern elements. To solve both problems, we consider an *occurrence* of a repeating pattern a *potential operation*. We then construct a snapshot of the pre- and post-memory states of the potential operation, and assign a label based on the difference between these. This labelling is performed by matching against a repository of *templates* for known data structure operations. Finally, the program's data structures can be identified by considering which operations manipulated which connected components in the points-to graphs.

We have implemented our proof-of-concept approach as a prototype [PSP1], which took approximately nine person-months to write. The current prototype employs user-specified templates to identify iterative non-nested data structures such as lists, queues, stacks, etc., in C programs. For evaluation targets we first considered textbook implementations of data structures and then real-world programs (see [PSP1] for references). The data structures in these programs were limited to linked lists, and queues and stacks implemented using linked lists. For both test domains, the identified operations were used to successfully differentiate between the different types of data structures, typically with 100% of identified operations supporting the correct data structure and only dropping from this (86% lowest) in three tests.

Approaches for Discovering Data Structures. The programming languages and compiler construction communities have proposed several approaches for discovering data structures within programs. The shape of a data structure is commonly abstracted by a *shape graph*, which permits finite represen-

tations of unbounded recursive structures. These may be discovered either statically [14] or dynamically; see [59] for use in profiling and optimization and [38] for use in detecting abnormal data structure behaviour. There also exist approaches that discover compound variable information from object code [22, 43, 44, 68], which are able to recover shape-graph-like representations.

The automated synthesis of *program invariants* [29] is also relevant for our work since it is possible to synthesize invariants that describe data structure shape. For example, the approach of [38] uses invariants based on the in/out-degree of shape graph vertices. When in/out-degrees begin to deviate from expected values, the invariants are violated and this can indicate programming errors. In contrast, *whole* heap invariants are constructed in the tool HeapMD [15]; however, these are more sensitive to non-relevant changes than invariants based on local structure. Given a set of example concrete structures, the tool Deryaft [49] extracts a set of invariants based on elementary graph properties, such as reachability and path length; this is extended to spectral properties on graphs in [48].

In contrast to the heap summarization approaches above, the dynamic analysis in [55] models the evolution of a complete Java heap allowing the collection of heap statistics such as object reachability and age. Reachability from entry points is used to perform a basic tree/DAG/cycle classification, similar in classification scope to the static approach of [30].

However, none of these approaches consider the operations affecting the data structures and, hence, fail to capture dynamic properties such as linked lists implementing queues. Furthermore, most abstraction techniques remove too much information about the data structures, e.g., some of the necessary concrete pointers, to be suitable for the type of analysis we wish to perform.

The tool DDT [39] is the most similar to the approach we propose in that it models the effects of operations on data structures, and is designed with the goal of optimizing data structure usage in application software [40]. DDT exploits the coding structure in standard library implementations, such as the Standard Template Library (STL) of C++ and the Gnome Library (GLib), to identify interface functions for data structures. Invariants are then constructed, describing the observed effects of an interface function, and these are used in turn to classify the data structure being manipulated. The reliance on well-structured interface functions means the approach is firmly targeted at modern application software, and would likely be unable to handle the ad-hoc interfaces appearing in OS/legacy software, or the scattered interfaces that can appear due to function inlining. In contrast, our machine learning approach makes fewer assumptions about the structure of the code that is used to manipulate data structures, and thus has much broader scope.

Approaches for Verifying Data Structure Usage. We now turn to related research on verifying correct data structure usage, which has been conducted by the automated verification community; this includes approaches based on logics, static analysis and graph transformation.

Separation logic permits local, logical reasoning about heap structures, enabling modular correctness proofs for pointer-manipulating programs [54, 61]. Tools such as SpaceInvader [73], Abductor [11], SLAyer [7], jStar [24] and VeriFast [36] are based on separation logic and have been used to verify a range of high- and low-level systems software, including Linux device drivers. *Shape analysis* [72] is another popular way to reason about the correctness of heap pointer structures. Based on a static analysis, it considers the formalization of shape graphs by three-valued logical structures [9, 65]. These may be obtained from pointer programs by predicate abstraction [33], so as to yield Boolean programs that conservatively preserve program behaviour [2, 23, 56]. Graph transformations may be used to describe the evolving shapes of heap structures by abstract graphs, and to implement pointer manipulations by graph transformation rules [60]. A grammar-based approach to heap abstraction is presented in [42]; however, it only supports tree data structures.

In [62, 63], a graph-based framework is advocated that allows for the verification of properties of heap-manipulating programs such as pointer and shape safety, pointer aliasing and termination. By combining *temporal logic model checking* with an abstraction framework based on *graph transformation*, the correct usage of data structures can be verified. However, the graph transformations for the data structure under analysis must be specified by hand and also be compatible with the property that one wishes to check. Tool support is provided by Juggrnaut [34] and this has been applied successfully to list and tree algorithms. The Verifying C Compiler (VCC) [18] provides an alternative approach to verification, by annotating C programs and then using verification condition generation as its underlying technology.

VCC has been employed in case studies on verifying low-level OS components and supports inlined assembly.

To configure these frameworks for specific programs, some information on the shape of the data structures under analysis must be known *a priori*, although there exists some work on inferring predicates automatically [33]. One of the motivations behind this project is to interface data structure discovery with verification frameworks to improve automation and scope of application.

Machine Learning & Pattern Matching. Our proof-of-concept approach for data structure identification applies basic techniques from machine learning and pattern matching. In order to handle more complex classes of data structures (e.g., with nesting) or different coding styles (e.g., recursion) required in the project proposed here, more advanced learning methods will be needed.

To capture the dynamic behaviour of a data structure we must model the behaviour of the operations that manipulate it, and then subsequently use these models for identification. The automatic construction of a model describing the behaviour of a process, e.g., a business process, is the goal of the discovery phase in *process mining* [71]. The input to construct this model is a set of event logs, each capturing the events that took place during one execution of the process. Both genetic algorithms and the minimum description length criterion have been employed for this task [12, 70]. The distinction between process mining and our own approach is that our input is not a *set* of executions of the operation, it is instead one continuous trace, where some sub-sequences of that trace correspond to executions of the operation. Thus, the set of executions of an operation and the models of operation behaviour must be learnt simultaneously, which is a significantly harder problem. As an aside, it is also common to model systems and processes with an automata-like formalism, and there exist some methods to automatically infer such models (see, e.g. [50]). Again, this is not easily applicable to our work since we have neither individual examples of an operation's behaviour nor negative examples to provide to a learner.

Due to the structure of our input data, we must employ different methods of learning the operations appearing in the trace, such as searching for repeated event sequences due to repeated operation invocations; this can be performed efficiently using suffix trees [67]. However, merely locating consecutive repeating sequences is insufficient to model the complex behaviour that operations will display; we need multi-level models to support nested control flow constructs or hierarchical operations on nested data structures, and generalization to model variation in operation execution. In [35], a method is described to locate approximate repeating patterns; however, the variation in the patterns is not modelled explicitly. In [37], it is shown how suffix trees can be used to find approximate repetition in a trace by learning multi-level consecutive and non-consecutive repetition. The work is presented in the context of process mining and is aimed at discovering low-level sub-processes contained in a set of logs. The disadvantage of this approach is that suitable parameters must be chosen, and in our application this may require *a priori* knowledge about the program. In contrast, our current process based on the minimum description length requires no such parameters.

Object Code Analysis. In order for our approach to be applicable to object code, the program semantics we employ when source code is available must be inferred from object code; for example, both pointer writes and compound variables must be identified. This is a very challenging problem; however, there exist approaches that allow the automated recovery of some of this information. Early such approaches were static and considered memory accesses very conservatively; they targeted the recovery of high-level expressions [16], control flow [8, 17] and type information [51]. Methods such as [64] allowed for a more accurate recovery but required the availability of debugging information. More recently, in [26], types were reconstructed for C programs by using a lattice over properties of data types. Essentially, these techniques are solving the problem of *decompilation*: the reverse engineering of a high-level language representation of a binary program (see, e.g., CodeSurfer/x86 [5]). In the following, we describe approaches specifically targeting the recovery of aggregate structures and type information as is required for our proposed approach (see [69] for a survey of these).

Divine [4] is a static analysis tool that recovers the syntactic structure of compound variables appearing in a binary. An abstract interpretation technique is described, which employs aggregate structure identification [58] and value set analysis (VSA) over abstract locations [3]. Memory is initially partitioned into global, stack and heap regions and then iteratively partitioned to discover the variables.

The following three dynamic approaches solve a similar problem and allow for the recovery of some semantic information. Laikia [22] is capable of recovering a type of shape graph describing the relations between compound variables, and these graphs are used as signatures for malware detection. REWARDS [44] performs type inference over timestamped memory locations. The type of a memory location (a variable) is resolved when it is involved in a type revealing operation, e.g., a system call, a library call or an instruction with type constraints on operands. Howard [68] allows the recovery of data structures from gcc-generated binaries by observing memory accesses and employing a code coverage tool to ensure that the executions used as input are suitably comprehensive. In contrast, the tool TIE [43] is applicable in both a static and dynamic setting. It begins by raising the object code to the authors' own representation which includes low-level typing information. Variable recovery is then performed on this representation by analysing access patterns in memory before finally performing type reconstruction by constructing constraint systems based on variable usage.

There are a number of instrumentation frameworks that can aid us in applying our approach, which currently uses the C Intermediate Language [52] for instrumenting C source code, to object code and in generating suitable program traces. For example, Valgrind [53] is commonly employed to build dynamic analysis tools; similar tools include PIN [46], Vulcan [28] and LLVM [41]. Disassembly can either be handled by the framework (e.g., converting to the Valgrind intermediate representation) or by a specialized tool such as IDApro [27].

As an important aside, the automated verification community is also starting to target the verification of object code. One example is the work described in [PSP2, PSP3] which considers not the verification of data structures, but the correct usage of pointers from the point of view of the operating system. The approach uses symbolic execution to update a custom model of memory, which may then be used to check correct (de)allocation, API usage rules and access writes. The application domain of this analysis is primarily OS software such as device drivers [PSP2] and virtual file systems [PSP3].

Obfuscation/Malware. A number of techniques can be used to obscure the meaning of a program, ranging from the fun hand-coded programs submitted to "The International Obfuscated C Code Contest" (`http://www.ioccc.org/`) to the low-level techniques of [45] for preventing static disassembly of object code. Since a program in binary form can often be correctly disassembled, our primary focus is on semantics-preserving transformations and dynamic obfuscation (see [20] for a survey of these).

On the one hand, semantics-preserving transformations are commonly used in program optimization; however, here the goal is to apply transformations that inhibit program comprehension. Static techniques such as control flow flattening, extra aliasing insertion [21], bogus control flow insertion and exploiting different data encodings all belong in this category. Also under this scope are transformations that break assumptions about program abstractions, such as splitting and merging classes.

On the other hand, dynamic techniques aim to modify or build the code at runtime [19]. For example, the technique in [1] modifies the program by splitting it up into pieces and then cyclically rearranging their placement during execution. The approach of [47] allows functions to share the same memory location; just before the function is called, the memory is manipulated to form the appropriate function. Finally, encryption may be used to hide the meaning of all but the small part of the program that is currently being executed (e.g., [13]).

These approaches and many others are employed in writing malware [66], a class of software which we will study in this project. We expect our approach to be resilient to a great number of these, since patterns in data structures are not common obfuscation targets [22]. Furthermore, even heavily obfuscated programs will leak some information in the form of repetitive memory access patterns, and our approach will be able to exploit this information.

1.1 Project-Related Publications

Reference [PSP1] details the concrete prior work that this proposal is based on. The references [PSP2, PSP3] describe Prof. Lüttgen's work on checking pointer safety properties on object code; the experiences here will be valuable for enabling our proposed approach on object code. Dr. White used pattern recognition techniques in [PSP6] to construct generative models over sets of graph, and later applied these to graphs representing chemical structures [PSP4]. The pattern recognition techniques employed here will aid us in recognising dynamic data structures represented as graphs. White and Lüttgen's joint

work [PSP5] is also relevant for this project as a significant focus was on interfacing different program representations via source code transformation; this is very similar to what we plan to do in this proposal in the instrumentation step.

1.1.1 Articles Published by Outlets with Scientific Quality Assurance

[PSP1] D. White and G. Lüttgen. Identifying Dynamic Data Structures by Learning Evolving Patterns in Memory. *Accepted for publication in the 19th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013). The proceedings will appear in Springer's Lecture Notes in Computer Science (LNCS) series. The final version of this paper is included with this proposal, together with the letter of acceptance.*

[PSP2] J. T. Mühlberg and G. Lüttgen. Symbolic object code analysis. *Intl. J. Softw. Tools Technol. Transfer*, Published online in August 2012. DOI: 10.1007/s10009-012-0256-8

[PSP3] J. T. Mühlberg and G. Lüttgen. Verifying compiled file system code. *Form. Asp. Comput.*, 24(3):375–391, 2012. DOI: 10.1007/s00165-011-0198-z

[PSP4] D. White and R. C. Wilson. Generative models for chemical structures. *J. Chem. Inf. Model.*, 50(7):1257–1274, 2010. DOI: 10.1021/ci9004089

[PSP5] D. White and G. Lüttgen. Embedded systems programming: Accessing databases from Esterel. *EURASIP J. Embedded Systems*, 2008(1):961036, 2008. DOI: 10.1155/2008/961036

[PSP6] D. White and R. C. Wilson. Spectral generative models for graphs. In *Intl. Conf. on Image Analysis and Processing* (ICIAP 2007), pp. 35–42. IEEE Computer Society, 2007. DOI: 10.1109/ICIAP.2007.119

2 Objectives and Work Programme

2.1 Anticipated Total Duration of the Project

The project has not yet started. Its duration shall be 36 months, for which DFG funding is sought.

2.2 Objectives

Aim. The project's aim is to design, implement and evaluate automatic techniques for learning information about the dynamic data structures employed by a pointer program from traces of its execution. In addition to identifying the data structures (data structure *shape*) we wish to also understand their operational *behaviour*, so as to enable a more accurate classification. For example, operational behaviour is required to recognize a linked list that is being used as a queue.

Our vision for this work is as follows. Firstly, the input program is automatically instrumented such that it produces a suitable trace when executed. For each trace generated, the analysis is performed to identify the data structures and associated operations. This information may be viewed by a GUI which presents the results in a user-friendly manner. Then, in an optional step, the information learnt from the analysis of *multiple* traces may be aggregated to provide a more comprehensive view of a program's behaviour. Finally, the analysis results are provided to external tools that may use the knowledge of data structure usage to improve their own analyses.

The key idea is to exploit the repetitive memory usage patterns produced by repeated invocations of a data structure operation to first locate and then identify the operation. The set of operations associated with a data structure can then be used to determine its type. By exploiting memory usage patterns, our approach is applicable at low levels of abstraction; thus, we aim to apply the approaches to be developed on object code, in addition to the goal of analysing source code.

There are challenging problems to be solved in this work at each level. Firstly, in order to model the data structure operations, the sub-sequences of the trace due to their invocation must be located. This is difficult because the trace will record the effect of both data structure operations and non-operations indiscriminately. Secondly, the located operations must be identified, and the *a priori* knowledge provided for this task must be both general enough to identify the operation in many different situations, and

specific enough to avoid generating false-positives. Thirdly, the identification of data structures from the set of operations that manipulate them is non-trivial since data structures may by split or joined over the course of program execution. Lastly, transferring the approach to object code level will be challenging as much of the required semantic information is missing in an object code representation, and even though external approaches can be employed to recover some of this information, it will generally be imprecise due to approximations.

The information learnt from such an analysis has significant application in multiple domains. Foremost, such information can be used to inform and guide formal verification techniques that require *a priori* information about a program. This can allow the tools to be applied to more complex pointer programs, for which it was not previously possible to analyse without user assistance. The approach also has application for program comprehension and is especially important when legacy software is considered. Many legacy programs contain bespoke implementations of data structures, and the understanding of these can be of great value when updating or maintaining the software. Finally, because data structure layout is not a common obfuscation target [20, 22] we expect the approach to be applicable to this type of software, thus allowing, for example, the analysis of malware.

Detailed Objectives. The concrete objectives of this project centre around the following themes, which will allow the application of our approach to the domains of OS software, application software and legacy/obfuscated software.

1. *Develop and evaluate machine learning and pattern matching algorithms to identify the dynamic data structures and operations typically used in device drivers (WP-2A/3A/4A).*

   Our existing proof-of-concept implementation [PSP1] shall form the basis of a framework for the development and evaluation of algorithms to handle the class of data structures and operation coding styles typically found in device drivers. Specifically, algorithms will be developed to handle nesting in data structures, particularly nested list structures, which are heavily used in device drivers. This objective only considers the analysis of device drivers for which source code is available, and not more general OS software since that typically relies on more complex data structures, often involving recursion. The approaches to be developed will be evaluated via case studies that also assess the usefulness of the analysis output in guiding verification tools, many of which target operating systems software [7, 36, 73], and particularly device drivers. Device drivers are a good choice for the first evaluation target as they are often written in a regular style which eases analysis, and are run in kernel-space where errors can be dangerous.

2. *Develop and evaluate machine learning and pattern matching algorithms to identify the dynamic data structures and operations used in general software (WP-2B/3B/4B/4C).*

   The algorithms developed for nested data structures will complement those developed for this objective to provide a unified approach for identifying data structures and operations occurring in software with available source code. Specific goals for this objective are to locate and identify recursively-coded operations, which will be of key importance in analysing programs that manipulate tree data structures. Case studies in all three domains (but excluding device drivers) will be conducted to assess the quality of the analysis output and to evaluate the ability of our approach to inform formal verification. For OS software, we will consider kernel components, e.g., the scheduler; for application/legacy software, examples will be taken from common/early Linux applications [11]; finally, obfuscated software examples will be generated through the use of techniques from [20].

3. *Adapt the algorithms and interface with external technologies to enable the analysis of object code (WP-8).*

   It is often necessary, or desirable, to perform the analysis on object code instead of source code, e.g., for programs with embedded assembler or verification methodologies based on executable code. Thus, a key goal of this objective is the selection, integration and evaluation of external technologies that can provide the necessary program semantics from the object code representation. A typical problem in trying to extract such information from object code is that the analysis method must make approximations, and this will result in imprecise information being used as input to our analysis. Therefore, the second goal of this objective is to develop and evaluate methods to ensure our machine learning and pattern matching algorithms are robust.

4. *Evaluate the analysis on object code (WP-9A/B/C).*

   The approach developed to handle the analysis of object code shall be thoroughly evaluated through case studies for each software domain of interest. In the domain of OS software, the primary evaluation target will be device drivers, in addition to some core components of an operating system, for example the scheduler. For application software, closed-source commercial off-the-shelf components will be mimicked by analysing a set of compiled open-source projects. Lastly, the obfuscated software domain will be covered by showing the tool is robust against common obfuscation techniques [20] such as those used in malware.

5. *Provide tool support and visualization of analysis results (WP-5/10).*

   For the approaches developed in this work to be adopted in practice, there must be a user-friendly tool that allows non-experts and analysis tools easy access to the output of our approach. Therefore, a major goal of this work is to provide a GUI that can present the analysis results to the user in a clear and instructive manner, and the development of an API that will allow other analysis tools to query the results of our analysis tool. The GUI will present the results in terms of the language of the input program (source or an assembler-like intermediate representation), in order to allow the user to easily relate the results to the code responsible. If the user requires a more detailed view of the effect of a data structure operation, then they will be able to view its effect on the heap in an informative visualization that also provides cross-referencing to the code view.

6. *Explore the limits of the approach (WP-6/7).*

   This objective covers two areas that are non-essential to the success of this research project but would nevertheless be of significant benefit. The first area concerns the development and evaluation of methods to aggregate analysis information from multiple traces (e.g., on the basis of coverage criteria). Clearly, for any reasonably sized program, a single trace cannot adequately explore all possible behaviours; thus, in order to build a comprehensive view of a programs behaviour, information from multiple traces must be combined. The second area involves utilizing the payload data contained in data structure elements to learn additional information about the data structure; for example, the location of the key element on which a data structure is sorted can enable the recognition of ordered data structures.

Outcome & Impact. The project's primary outcome will be a novel approach employing machine learning and pattern matching algorithms to automatically identify and label data structures and their associated operations from traces of automatically instrumented executions of pointer programs. Tool support will be provided for the approach, thus making it accessible to non-experts and allowing its interfacing to other program analysis tools.

Providing high-quality information on shape, operational behaviour and coding style of a pointer program's data structures will be of significant help for programmers as well as various automated analysis tools. We expect our approach to be especially beneficial to engineers working to maintain legacy systems, in addition to engineers performing reverse engineering. Equally importantly, the analysis results can also be used to inform those program analysis and verification tools that rely on a knowledge of the data structures employed in a program. This will enable such tools to automatically handle a class of programs that are currently only analysable with user assistance. Moreover, a high-level understanding of the data structures used by a program can aid debugging/profiling/optimization tools and enable the creation of program signatures for security.

2.3 Work Programme including Proposed Research Methods

The proposed programme of work involves 17 work packages. The scheduling of all work packages, including the person who will undertake the work and the involvement of our international partners at the University of York, UK, and KU Leuven, Belgium, is depicted in Figure 1. The work packages are structured such that those concerning one software domain minimally interfere with those for the other two domains; furthermore, WP-6/7 are re-schedulable independently of the other packages. Dr. White will spend approximately 50% of his full-time position on the project until his existing contract as Post Doc (Wissenschaftlicher Mitarbeiter) at the University of Bamberg finishes in November 2015. From this

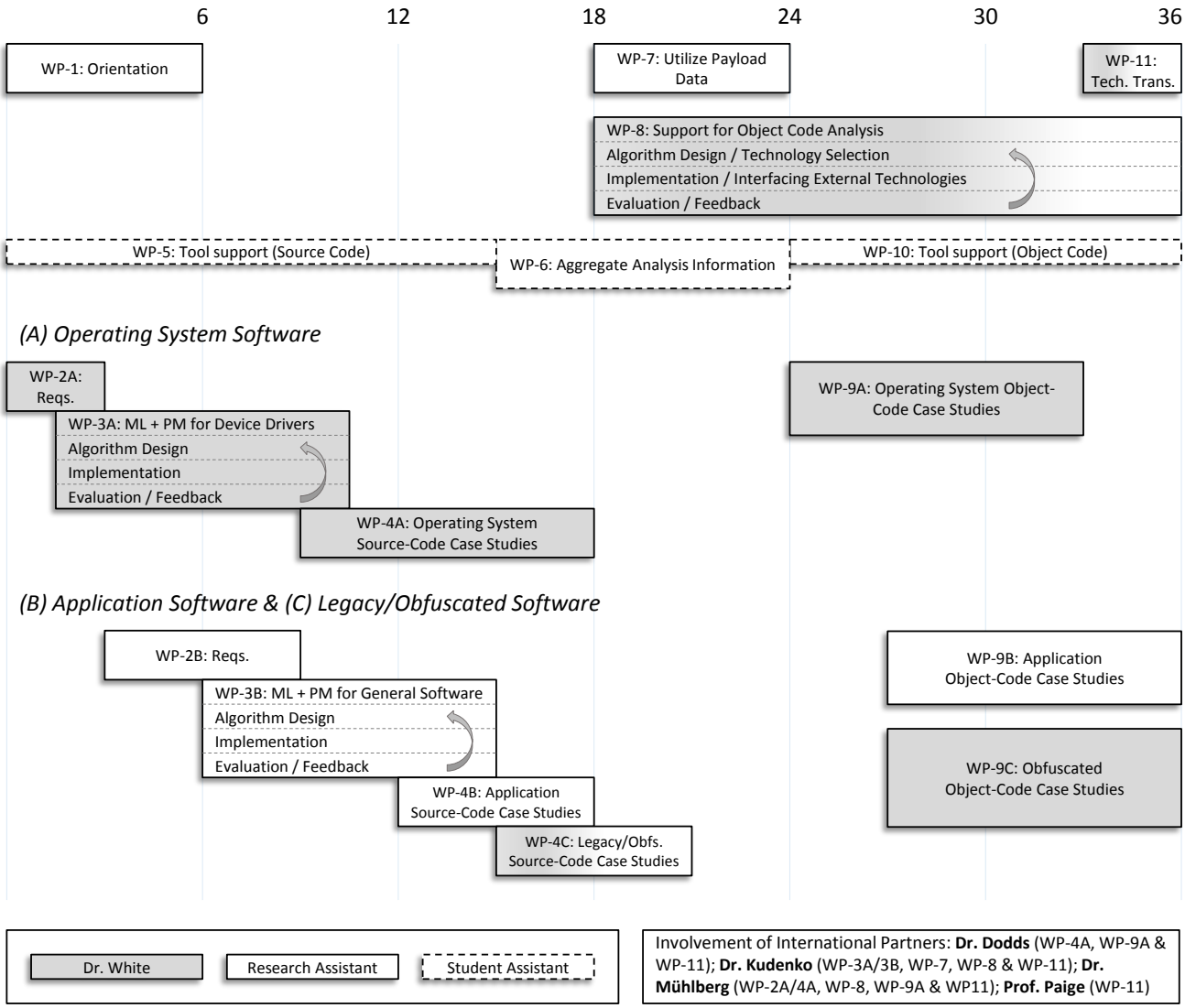

Figure 1: Scheduling of work packages.

point on he will be employed to work on the project full-time until the project's conclusion in September 2016. The Research Assistant to be hired will work full-time time on the project, and the Research Student will work on a wages-per-week basis of 10 hours per week on average.

Funds are required to cover the salaries of the Research Assistant and the Research Student for the full duration of the project, and White's salary for the final 10 months of the project. Some work packages will be conducted with advice from our international partners: Dr. Dodds, Dr. Kudenko and Prof. Paige (University of York) as well as Dr. Mühlberg (KU Leuven).

WP-1: Orientation. The project shall start off for the hired Research Assistant by familiarising himself/herself with the relevant literature in the field. In addition, the Research Assistant will spend some time working with the existing prototype tool and studying the examples already evaluated to gain a good understanding of the current state of our approach.

WP-2A/B: Software Domain Requirements. The goal of these work packages is to compile a set of requirements that our tool will have to fulfill in order to analyse traces of device drivers (WP-2A), application software (WP-2B) and legacy/obfuscated software (WP-2B), when the source code of the program under analysis is available. To gather these requirements, a representative set of programs from each of these domains will be collected and then analysed by hand. Our partner Dr. Mühlberg will assist us with the collection of device driver examples. Furthermore, our proof-of-concept implementation will be employed in its current state to better understand its limitations.

Clearly, there will be some overlap in the requirements between the three domains. For example, all three domains will require that the analysis can handle nested data structures. However, the analysis of tree data structures and the associated standard implementation methods utilizing recursion do not

feature heavily in device drivers. Instead, device drivers are primarily composed of nested list structures where some lists implement stacks and queues. We expect that generally the requirements for device drivers will form a subset of those for application and legacy software, and therefore the technology produced for the analysis of device drivers will be a basis for the other domains. This observation allows the work to be decomposed between two researchers: Dr. White will undertake the research on device drivers (WP-2A), while the hired Research Assistant will perform the research on application and legacy/obfuscated software (WP-2B). This division allows WP-2A, and the realisation of those requirements in WP-3A, to be initiated while the Research Assistant is familiarizing themselves with the project in WP-1. Prof. Lüttgen will also contribute to WP-2A/B.

WP-3A/B: Analysis Technology (Source Code Available). In these work packages we will employ an iterative development approach to implement solutions for the requirements identified in WP-2A and WP-2B. The iterative approach will consist of designing machine learning and pattern matching algorithms, implementing these in the prototype tool and evaluating their accuracy. The results of the evaluation will feed back to the next phase of algorithm design. As before, the work for analysing device drivers (WP-3A) will be performed by Dr. White, and the work for application and legacy/obfuscated software (WP-3B) will be performed by the Research Assistant. We will consult with our partner Dr. Kudenko regarding the technologies to be developed here; Prof. Lüttgen will also be involved.

A key challenge in WP-3A will be extending the current machine learning and pattern matching approaches to locate and identify operations on nested data structures. One way to accomplish this is to allow the learning of more expressive patterns by the genetic algorithm. For this to work, it is likely that changes will be necessary to the features to expose nested repetition. Furthermore, the types of templates employed in the template matching step must be made more flexible, likely through a compositional approach that allows a template to match each level of the nested data structure.

The notion of entry points to a data structure will also be of increased importance; these can now originate from a standalone pointer or an element of another data structure. In fact this is currently a problem when we wish to identify locations in symmetric data structures, e.g., when identifying the ends of a doubly linked list with pointers to both the head and tail. Clearly, the consistent identification of list ends is a requirement to recognize lists implementing stacks and queues. A possible solution to this is to allow successful template matches to impart information on the points-to graph that may be used by subsequent template matches, such as marking list ends with unique identifiers.

We expect the key challenges of WP-3B to be the sheer variety of data structures utilized in application and legacy software. In this regard, we must be careful to limit the scope to data structures that we can reasonably hope to analyse. We will primarily focus on linked lists (and the subclasses of these) and tree data structures. In work package WP-7 we will consider extensions to ordered subclasses of these, such as balanced trees. Regarding obfuscated code, we expect the challenge to lie in making our approach robust against the introduction of noise in the program trace due to the obscuring transformations.

The different coding styles utilized to implement these data structures will present added challenges. For example, trees are commonly implemented using recursion, and the repeating patterns in the program trace caused by recursion are very different to those caused by iteration. This is due to the tendency of recursive calls to break the execution of a function into smaller non-contiguous sections. Improvements will need to be made to the machine learning step to handle this type of pattern.

To summarise, there are a number of paths to pursue in WP-3A/B, which exploit and extend our current machine learning and pattern matching framework. For both work packages we expect the key to success is for tighter integration between (a) the learners and (b) the representations of the program trace on which the analysis is performed. An example of (a) would be including a template matching criteria in the genetic algorithm's fitness function. This allows learnt patterns to be "frozen" once they correctly match a known operation and, thus, the complexity of the learning problem is reduced. An example of (b) is to exploit pointer write locality in data structure operations (expressed in the points-to trace) for locating the beginning/ends of operations (performed using the feature trace). Lastly, we note that a tighter coupling between the abstractions used to represent the trace will help one to alleviate the assumptions introduced earlier, namely that we require a large number of invocations of an operation and that the context surrounding those invocations should vary.

WP-4A/B/C: Case Studies and Technology Transfer (Source Code Available). In these work packages, the new approaches from WP-3A/B will be evaluated through case studies for each domain. In contrast to the purely accuracy-based evaluation utilized in WP-3A/B to provide algorithm design feedback, here we will perform a more comprehensive evaluation focusing on two areas.

The first type of case study will be a large scale evaluation of the quality of information learnt by the approach, so as to access the usefulness of the approach for program comprehension. A simple example of a measure we could use for this would be how accurately the lines of code corresponding to the learnt operations relate to the true lines of code for that operation. This information will be especially useful when analysing legacy software or when a purely source-based obfuscation process has been performed. The second type of case study will evaluate the usefulness of the tool's output to inform formal verification techniques. For example, knowing what data structures one is dealing with would help one to instantiate the shape analysis framework with more relevant, better fitting predicates, thereby increasing the precision of the analysis and verification. Furthermore, information on the locations of data-structure operations in the code would allow one to "fast-forward" an analysis over uninteresting program parts, thereby making verification more efficient.

In WP-4A, Dr. White will perform case studies on a set of device drivers that contain no embedded assembly code. This set can be collected from the shape analysis literature since most approaches there are also not applicable on code with embedded assembly (see Section 1). By using examples from the shape analysis literature we will be able to more accurately determine the usefulness of our tool to inform verification tools such as [7, 36, 73]. Our partners Dr. Dodds and Dr. Mühlberg will be able to provide guidance for this. When significant progress has been made on WP-3B, there will be an option to evaluate the approach on non-device-driver OS components, such as schedulers which often take the form of tree data structures.[1] It is for this reason that the duration of this work package (WP-4A) is set slightly longer than the others (WP-4B/C).

The Research Assistant will perform the evaluation for the domains of application software (WP-4B) and will work together with Dr. White on legacy/obfuscated software (WP-4C). Representative examples of application software will be borrowed from active open-source projects written in C, e.g., Gimp, Putty and GNUplot. Further examples will be taken from [11] to enable comparison with the Abductor tool. Examples of legacy code will be collected from early versions of popular Unix applications, such as sh, emacs, vi, grep and sed. A preliminary study on obfuscation is also possible here since there exist many algorithms to obscure source code (see, e.g. [20]).

WP-5: Tool Support for Analysis Output (Source Code available). This work package concerns the construction of a GUI, visualization methods and tool support to present the results of the analysis to the user in an instructive way. In addition, an API will be developed to allow other analysis tools to query the data structure information learnt by our approach. Lastly, to aid development of the machine learning and pattern matching algorithms, the GUI will provide a number of views of the intermediate results produced by these algorithms, in order to enable their evaluation. It is essential to have this type of view available to developers since the traces are simply too large to be handled manually. This work package occupies quite a long period of time because it is expected that the GUI/API will not be in constant development, but instead will be updated in response to requirements from the development (WP-3A/B) and evaluation (WP-4A/B/C) work packages. This work will be performed by the Student Assistant under the guidance of the Research Assistant and Dr. White.

Because the GUI will be utilized for two different purposes, it will provide two different views for these usage scenarios. In user mode, the main view will be the source code of the project, and overlaid on this will be the information learnt by the tool. Examples of overlaid information are highlighted lines of code that comprise labelled data structure operations, and variables that represent entry points to labelled data structures. In addition, the user will be able to "step through" the lines of code that comprise an operation and see this played out on a visualization (points-to graph) of the heap. Finally, information will be cross-referenced between the code view and the visualization.

WP-6: Aggregating Information over Multiple Traces. In this work package we will consider how a more complete view of a program's data structure usage can be obtained through the analysis of

[1] `http://www.ibm.com/developerworks/linux/library/l-cfs/`

multiple traces. If the traces are describing different parts of program operation, then the analysis results can simply be combined. However, the more interesting case is to decide how information should be aggregated when it concerns the same aspect of the program. A simple example of this information aggregation would be to improve the confidence on a code region labelled as a data structure operation, by checking that the region receives the same operation label in all analyses. A similar check can be performed on variables representing entry points to a labelled data structure.

To generate a useful set of traces that adequately explore a program's behaviour will require a code coverage criterion (e.g., [10]) as is used in software testing. Therefore, the location and integration of such a technology will be the first task for this work package. After this, the design, implementation and evaluation of information aggregation strategies will be performed, and the GUI will be updated to relate this aggregated information to the user. Finally, a case study will be conducted on a number of example programs, like those analysed in WP-4A/B/C, which will compare the quality and usefulness of the learnt information when one trace is used and when multiple traces are used. This work will be carried out by the Student Assistant under the guidance of the Research Assistant. The timing of this work package is flexible; it may be performed at any point after WP-3A/B is complete. Thus, rescheduling is possible if work on the other packages proceeds faster or slower than expected.

WP-7: Utilizing Payload Data. In this work package we will investigate the possibility of using the data stored in a data structure (the *payload data*) to improve the identification accuracy. The obvious use case for this type of analysis is to identify data structures where an ordering is imposed over the elements. By employing machine learning, domain knowledge and carefully observing the position of element inserts, we expect that the key on which a data structure's order is determined can be discovered. This approach might also be successful in identifying "meta" elements of data structures such as header or sentinel nodes, where the values of the payload determine the role that the element plays. This work will first be evaluated on ordered data structure examples taken from textbooks and then tested on those real-world programs utilized in WP4-A/B/C that use ordered structures. This work will be performed by the Research Assistant, and the timing is similar to WP-6 in that it may be scheduled at any point after WP-3/B is complete. Our machine learning partner, Dr. Kudenko, will be consulted in relation to this work package.

WP-8: Analysis Technology (Object Code Available). Up until this point, it has been assumed that the source code for the program under analysis is available. This enables a very precise instrumentation to be performed and, in turn, the information contained in the program trace is of a high quality and is suitable to construct the abstractions that our approach relies on. In this work package we consider what changes will be necessary to analyse programs for which the source code is not available, i.e., when we only have a program binary. Essentially, we will perform a preprocessing step on the program binary using static/dynamic tools capable of recovering information relevant for our approach, and then use this information to proceed with our dynamic analysis as before.

The first major challenge is identifying the program events of interest; to recap we require the logging of writing pointers to memory, memory allocation events and scoping information for all temporary pointer variables. Memory allocation events are deferred to system calls, so these will be no problem. Pointer write events are harder as we must determine which memory writes constitute pointers. Finally, scoping information is quite difficult since this relies on tracking the behaviour of the runtime environment. The second major challenge is the construction of the points-to graph. For this, we must compute which memory addresses belong to the same compound variable, so that this range of addresses can be correctly represented by a single vertex in the points-to graph.

To solve both these challenges we will take our inspiration from technologies described in related work (e.g., [4, 22, 43, 44, 68]). These technologies fall into two main groups: aggregate structure identification, which helps determine the memory address ranges that are data structure elements, and type inference, which allows types to be determined, and thus the identification of pointer writes. However, the automated deduction of scoping information will be harder, and we may have to rely on domain knowledge of the runtime environment in use.

Observe that none of these object code analysis techniques gives precise results for real-world programs, since there is often insufficient information available in the object code. Imprecision itself is not a significant problem for our approach; it simply means that the results of our analysis will also be

imprecise. Instead, the problem occurs when imprecision results in conflicting information being generated. An example of this could be that the estimated address range of a compound variable changes as the program executes. Conflicts such as this must be resolved; in this case, it could be addressed by allowing vertices in the points-to graph to be merged or split.

Lastly, note that this analysis will not be carried out directly on the object code; we will perform a disassembly preprocessing step to transform the object code into a suitable intermediate language (e.g., Valgrind [53]) on which the analysis will take place. This generalizes the approach and will allow the analysis of object code from different architectures. Thus, part of this work package will be the selection of a suitable intermediate language and associated disassembler.

This work package will be undertaken using an iterative methodology. The first phase will consist of technology selection and adaptation, in addition to making the machine learning and pattern matching algorithms robust against imprecise information. In the second phase, the algorithms will be implemented or updated and interfaced to the selected external technology. Finally, the success of the tool in its then current state will be evaluated, and this information will feed back and influence the next cycle. This work package will be undertaken by both Dr. White (75%) and the Research Assistant (25%). We will consult with Dr. Kudenko for the machine learning components and with Dr. Mühlberg and Prof. Lüttgen on object code analysis.

WP-9A/B/C: Case Studies and Technology Transfer (Object Code Available). This work concerns the evaluation of the approach implemented in WP-8 on a variety of object code examples. We will perform case studies for each domain, and also consider how our tool can be interfaced with external technologies from the verification domain. In WP-9A, our tool will be evaluated on object code from device drivers and OS components; in WP-9B, object code of application software will be considered, and in WP-9C object code resulting from obfuscated software will be investigated. The results obtained will be compared to those from WP-4A/B/C, and thus we can directly relate the effectiveness of our approach when source code is available and when it is not.

WP-9A. Being able to analyse the object code of OS components improves on our previous ability to analyse OS source code considerably. OS software typically has some embedded assembler, but it is also common for proprietary components to be released only as binaries. Moreover, an analysis at object code level removes all reliance on the correctness of the compiler and its optimizations. Non-device-driver OS components will be taken from Windows/Linux, or smaller OSs (such as Minix), while the device driver examples will be taken from the literature (e.g., [6]). We will construct and perform experiments to show that the output of our tool can provide useful information to formal verification techniques targeting object code [PSP2]. This work will be performed by Dr. White, with advice from Dr. Dodds and Dr. Mühlberg.

WP-9B. Analysis targets for the approach in the application software domain include any pre-compiled pieces of application software, such as commercial off-the-shelf (COTS) components. However, without an understanding of the internals of such software it will be very difficult to evaluate how successful our approach is. Therefore, we will emulate this domain by using compiled examples from the open-source projects used in WP-4B. Again, we will assess how useful our tool's output is in (a) informing formal verification and (b) program comprehension. This work will be performed by the Research Assistant, with advice from Dr. Dodds and Dr. Mühlberg.

It is expected that application software will be the easiest of the three evaluation domains. This is because the object code will be exactly that resulting from a compiler, so that we can make use of some domain knowledge concerning typical code generation patterns of compilers. If the compiler is performing heavy optimization on the code, then the patterns might be unintentionally obscured and, in the worst case, heavily optimized machine code might belong to the obfuscation domain. The domain knowledge that we might be able to leverage concerns, e.g., control flow constructs that can guide the construction of suitable patterns for an operation, since we need no longer guess if iteration or branching is occurring. More abstractly, we may make assumptions about the call/return structure generated by a particular compiler and the way compound variables are laid out in memory.

WP-9C. Finally, in this work package, the performance of our tool on object code resulting from obfuscated software will be evaluated. The scope of our approach must be carefully defined here; we will

assume that it is possible to extract a correct representation of the program in whichever intermediate representation we choose. In other words, we will not consider the disassembly problem and instead leave this to other approaches, see e.g. [27]. With that said, we still expect the recovered intermediate representation to be very difficult to understand (i.e., still obfuscated).

This will clearly be the hardest evaluation target of the three discussed in this section because we can make no assumptions about the structure of the code in the intermediate representation or the runtime environment that maintains it. However, data structure layout is not a typical obfuscation target [20, 22], and therefore we expect the approach to handle common types of obfuscated software. Furthermore, our tool makes few assumptions about high-level control flow; for example, we do not require that the same code is always executed when control reaches a certain point. This should allow the analysis of programs that display polymorphic behaviour.

We will begin by assessing the tool's performance on pieces of application software that have been obfuscated using standard techniques described in [20]. Then, we will analyse our approach on samples of malware that are documented so we can validate our results. These malware examples will be taken from textbooks [66] and, if possible, we will analyse real-world samples. We anticipate that results from our approach could be integrated with malware visualization tools such as VERA [57]. This work will be performed by Dr. White, with advice from Dr. Mühlberg.

WP-10: Tool Support for Analysis Output (Object Code available). In this work package, the GUI, visualization methods, API and tool support developed in WP-5 will be extended to work correctly with the chosen intermediate representation. Of course, if both the source code and a mapping from the object to source level exists (e.g., via debugging information), then it will be possible to display the results obtained on object code in terms of the source code. When this is not possible, the main user view will move from displaying the source code for the program to displaying the intermediate representation. It will also be more complex to display the data structure entry points to the user, since the variables typically representing these will no longer be visible.

Furthermore, as it is quite possible that mistakes can now be made in the analysis due to incorrect interpretation of the object code, the confidence of certain choices made by the tool will be displayed, and the user will be given the opportunity to correct these by hand to some degree. The developer view will also be updated to be compatible with algorithms used for the object-code based analysis. Again, the Student Assistant will perform this work as needed in response to the requirements of WP-8/9, under the guidance of the Research Assistant.

WP-11: Technology Transfer to other Domains. In this work package we will consider how our approach can be applied in other domains, for example, to aid high-level profiling, similar to the ideas of [40, 59] and in the security domain by, e.g., constructing data structure/program signatures [22, 25]. We will discuss with experts in these fields and see if our approach can be of benefit; Prof. Paige's experience in the security domain will be of particular benefit to us. This work will be performed by Prof. Lüttgen, Dr. White and the Research Assistant with collaboration from our international partners.

2.7 Information on Scientific and Financial Involvement of International Cooperation Partners

Not applicable; see Section 5.4.1 for information on international cooperation partners with whom we will be working together informally. These partners neither have applied nor are planning to apply for funding with the DFG or a partner organization in the context of this project proposal.

3 Bibliography

[1] D. Aucsmith. Tamper resistant software: An implementation. In *IWIH 1996*, vol. 1174 of *LNCS*, pp. 317–333. Springer, 1996.

[2] I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *VMCAI 2005*, vol. 3385 of *LNCS*, pp. 164–180. Springer, 2005.

[3] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC 2004*, vol. 2985 of *LNCS*, pp. 5–23. Springer, 2004.

[4] G. Balakrishnan and T. Reps. DIVINE: DIscovering Variables IN Executables. In *VMCAI 2007*, vol. 4349 of *LNCS*, pp. 1–28. Springer, 2007.

[5] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86 – A platform for analyzing x86 executables. In *CC 2005*, vol. 3443 of *LNCS*, pp. 250–254. Springer, 2005.

[6] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV 2007*, vol. 4590 of *LNCS*, pp. 178–192. Springer, 2007.

[7] J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory safety for systems-level code. In *CAV 2011*, vol. 6806 of *LNCS*, pp. 178–183. Springer, 2011.

[8] J. Bergeron, M. Debbabi, M. Erhioui, and B. Ktari. Static analysis of binary code to isolate malicious behaviors. In *WETICE 1999*, pp. 184–189. IEEE, 1999.

[9] D. Beyer, T. Henzinger, and G. Théoduloz. Lazy shape analysis. In *CAV 2006*, vol. 4144 of *LNCS*, pp. 532–546. Springer, 2006.

[10] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation*, pp. 209–224. USENIX Association, 2008.

[11] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, 2011.

[12] T. Calders, C. Günther, M. Pechenizkiy, and A. Rozinat. Using minimum description length for process mining. In *SAC 2009*, pp. 1451–1455. ACM, 2009.

[13] J. Cappaert, N. Kisserli, D. Schellekens, and B. Preneel. Self-encrypting code to protect against analysis and tampering. In *1st Benelux Workshop on Information and System Security*, 2006.

[14] D. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *PLDI 1990*, pp. 296–310. ACM, 1990.

[15] T. Chilimbi and V. Ganapathy. HeapMD: Identifying heap-based bugs using anomaly detection. In *ASPLOS 2006*, pp. 219–228. ACM, 2006.

[16] C. Cifuentes and A. Fraboulet. Interprocedural data flow recovery of high-level language code from assembly. Technical report 421, Univ. Queensland, 1997.

[17] C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation. In *ICSM 1998*, pp. 228–237. IEEE, 1998.

[18] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying Concurrent C. In *TPHOLs 2009*, vol. 5674 of *LNCS*, pp. 23–42. Springer, 2009.

[19] F. Cohen. Operating system protection through program evolution. *Comput. Secur.*, 12(6):565–584, 1993.

[20] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley, 2009.

[21] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *POPL 1998*, pp. 184–196. ACM, 1998.

[22] A. Cozzie, F. Stratton, H. Xue, and S. King. Digging for data structures. In *OSDI 2008*, pp. 255–266. USENIX Association, 2008.

[23] D. Dams and K. Namjoshi. Shape analysis through predicate abstraction and model checking. In *VMCAI 2003*, vol. 2575 of *LNCS*, pp. 310–323. Springer, 2003.

[24] D. Distefano and M. Parkinson. jStar: Towards practical verification for Java. *SIGPLAN Not.*, 43(10):213–226, 2008.

[25] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *CCS 2009*, pp. 566–577. ACM, 2009.

[26] E. Dolgova and A. Chernov. Automatic reconstruction of data types in the decompilation problem. *Program. Comput. Soft.*, 35:105–119, 2009.

[27] C. Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, 2008.

[28] A. Edwards, A. Srivastava, and H. Vo. Vulcan binary transformation in a distributed environment. Technical report, MSR-TR-2001-50, Microsoft Research, 2001.

[29] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1–3):35–45, 2007.

[30] R. Ghiya and L. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL 1996*, pp. 1–15. ACM, 1996.

[31] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[32] P. Grünwald. *The Minimum Description Length Principle*. MIT Press, 2007.

[33] B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. *SIGPLAN Not.*, 42(6):256–265, 2007.

[34] J. Heinen, T. Noll, and S. Rieger. Juggrnaut: Graph grammar abstraction for unbounded heap structures. In *TTSS 2009*, vol. 266 of *ENTCS*, pp. 93–107. Elsevier, 2009.

[35] J.-L. Hsu, A. Chen, and H.-C. Chen. Finding approximate repeating patterns from sequence data. In *ISMIR 2004*, pp. 246–249, 2004.

[36] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NFM 2011*, vol. 6617 of *Lecture Notes in Computer Science*, pp. 41–55. Springer, 2011.

[37] R. Jagadeesh Chandra Bose and W. van der Aalst. Abstractions in process mining: A taxonomy of patterns. In *BPM 2009*, pp. 159–175. Springer, 2009.

[38] M. Jump and K. McKinley. Dynamic shape analysis via degree metrics. In *ISMM 2009*, pp. 119–128. ACM, 2009.

[39] C. Jung and N. Clark. DDT: Design and evaluation of a dynamic program analysis for optimizing data structure usage. In *MICRO 2009*, pp. 56–66. ACM, 2009.

[40] C. Jung, S. Rus, B. Railing, N. Clark, and S. Pande. Brainy: Effective selection of data structures. *SIGPLAN Not.*, 47(6):86–97, 2011.

[41] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO 2004*, pp. 75–86. IEEE, 2004.

[42] O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP 2005*, vol. 3444 of *LNCS*, pp. 124–140. Springer, 2005.

[43] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled reverse engineering of types in binary programs. In *NDSS 2011*. The Internet Society, 2011.

[44] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *NDSS 2010*. The Internet Society, 2010.

[45] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS 2003*, pp. 290–299. ACM, 2003.

[46] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI 2005*, pp. 190–200. ACM, 2005.

[47] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere. Software protection through dynamic code mutation. In *ISA 2006*, vol. 3786 of *LNCS*, pp. 194–206. Springer, 2006.

[48] M. Malik and S. Khurshid. Dynamic shape analysis using spectral graph properties. In *ICST 2012*, pp. 211–220. IEEE, 2012.

[49] M. Malik, A. Pervaiz, E. Uzuncaova, and S. Khurshid. Deryaft: A tool for generating representation invariants of structurally complex data. In *ICSE 2008*, pp. 859–862. ACM, 2008.

[50] M. Merten, F. Howar, B. Steffen, S. Cassel, and B. Jonsson. Demonstrating learning of register automata. In *TACAS 2012*, vol. 7214 of *LNCS*, pp. 466–471. Springer, 2012.

[51] A. Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *ESOP 1999*, vol. 1576 of *LNCS*, pp. 208–223. Springer, 1999.

[52] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, vol. 2304 of *LNCS*, pp. 213–228. Springer, 2002.

[53] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[54] P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *POPL 2004*, pp. 268–280. ACM, 2004.

[55] S. Pheng and C. Verbrugge. Dynamic data structure analysis for Java programs. In *ICPC 2006*, pp. 191–201. IEEE, 2006.

[56] A. Podelski and T. Wies. Boolean heaps. In *SAS 2005*, vol. 3672 of *LNCS*, pp. 268–283. Springer, 2005.

[57] D. Quist and L. Liebrock. Visualizing compiled executables for malware analysis. In *VizSec 2009*, pp. 27 –32, 2009.

[58] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *POPL 1999*, pp. 119–132. ACM, 1999.

[59] E. Raman and D. August. Recursive data structure profiling. In *MSP 2005*, pp. 5–14. ACM, 2005.

[60] A. Rensink. Canonical graph shapes. In *ESOP 2004*, vol. 2986 of *LNCS*, pp. 401–415. Springer, 2004.

[61] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS 2002*, pp. 55–74. IEEE, 2002.

[62] S. Rieger and T. Noll. Abstracting complex data structures by hyperedge replacement. In *ICGT 2008*, vol. 5214 of *LNCS*, pp. 69–83. Springer, 2008.

[63] S. Rieger. *Verification of Pointer Programs*. PhD thesis, RWTH Aachen University, 2009.

[64] X. Rival. Abstract interpretation-based certification of assembly code. In *VMCAI 2003*, vol. 2575 of *LNCS*, pp. 41–55. Springer, 2003.

[65] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

[66] M. Sikorski and A. Honig. Practical malware analysis. No Starch Press, 2012.

[67] S. Skiena. *The Algorithm Design Manual*. Springer, 2nd edition, 2008.

[68] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS 2011*. The Internet Society, 2011.

[69] X.-X. Sun, H. Chen, Y. Wen, and M. Huang. Reversing engineering data structures in binary programs: Overview and case study. In *IMIS 2012*, pp. 400–404. IEEE, 2012.

[70] W. van der Aalst, A. Alves de Medeiros, and A. Weijters. Genetic process mining. In *ICATPN 2005*, vol. 3536 of *LNCS*, pp. 48–69. Springer, 2005.

[71] W. van der Aalst. *Process Mining*. Springer, 2011.

[72] R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *CC 2000*, vol. 1781 of *LNCS*, pp. 1–17. Springer, 2000.

[73] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *CAV 2008*, vol. 5123 of *LNCS*, pp. 385–398. Springer, 2008.

4 Requested Modules/Funds

4.1 Basic Module

4.1.1 Funding for Staff

- *1 Research Assistant* (PhD student), for 36 months full-time (Doktorandin/Doktorand und Vergleichbare, 100% der regelmäßigen Arbeitszeit)

  The Research Assistant will work full-time under the supervision of the Prof. Lüttgen and Dr. White, taking up his or her position with the start of the grant. After using WP-1 to orientate him or herself, he or she will be responsible for the work packages concerning the analysis of general software (WP-2B/3B/4B), utilizing payload data (WP-7) and the case studies for analysing the object code

of application software (WP-9B). In addition, he or she will work with White on the case studies for legacy/obfuscated software at source code level (WP-4C), transferring the approach to object code level (WP-8) and the transfer of technology to other domains (WP-11). The split of work between the Research Assistant and White will be approximately equal on the shared work packages, with the exception of WP-8, where White will perform 75% of the work. The Research Assistant will also be responsible for guiding the Student Assistant while they are performing WP-5/6/10.

This post is full-time not only because of the proposed work load, but it is also necessary to attract suitable candidates, given the excellent job prospects in and the high starting salaries offered by the IT industry. The post will be advertised nationally and internationally if no suitable candidate among the graduates of Lüttgen can be recruited.

- *1 Postdoctoral Researcher*, for 10 months full-time (Postdoktorand, 100% der regelmäßigen Arbeitszeit)

  Dr. White will work part-time (approximately 50%) on the project (WP-2A/3A/4A/4C) until his current contract at the University of Bamberg (Akademischer Rat a. Z.) finishes on 30th November 2015. From this point on, funds are requested to allow him to work full-time on the project until its conclusion (a duration of 10 months). During this time, White will work on WP-8 (performing approximately 75% of the work) and WP-11 (performing approximately 50% of the work), in collaboration with the Research Assistant. He will also perform the case studies that comprise WP-9A and WP-9C. White is a co-author of this grant proposal, and his continued involvement in the project will be essential for its success.

- *1 Student Assistant* (Studentische Hilfskraft), for 44 hours per month for 36 months, at the standard rate of 14.08 Euro per hour, for a total of 22,302.72 Euro.

  The Student Assistant will be responsible for the GUI, API, visualization and packaging of the developed tool for end users (WP-5/10). In addition, he or she will perform the work on aggregating analysis information over multiple runs and presenting this to the user (WP-6). The Research Assistant will supervise the work undertaken by the Student Assistant. The Student Assistant will likely be recruited among the students who take Prof. Lüttgen's modules on software engineering and compiler construction.

4.1.2 Direct Project Costs

4.1.2.1 Equipment up to 10,000 Euro, Software and Consumables. Specialised equipment for carrying out the proposed project is not required and no funds for consumables are requested. All staff on the project will be equipped by the University of Bamberg with *state-of-the-art PCs and/or laptops* as well as *headsets and webcams* for communication with our international project partners. Moreover, all software necessary for carrying out this project is available free of charge.

4.1.2.2 Travel Expenses. Funds are requested to cover the cost of travel:

- *Conferences/workshops*

  The Research Assistant, Dr. White and Prof. Lüttgen will actively participate in international conferences/workshops during project. Funding is requested for the Research Assistant to be able to attend two such events per year and for each Dr. White and Prof. Lüttgen to attend one. It is expected that the Research Assistant will attend one event in Europe and one overseas, while White and Lüttgen will attend events in Europe.

  Examples of targeted meetings are, in alphabetical order, the international conferences on *Computer Aided Verification* (CAV), *Fundamental Approaches to Software Engineering* (FASE), *Foundations of Software Engineering* (FSE), *Principles of Programming Languages* (POPL), *Program Comprehension* (ICPC), *Programming Language Design and Implementation* (PLDI), and *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS), plus some of their affiliated, high-quality workshops.

  The cost of a European trip is estimated at 1,500 Euro (including registration fees, accommodation and travel expenses), and the one for an overseas trip at 2,000 Euro. Therefore, 19,500 Euro is requested in total for all conference/workshop travel.

- *International cooperation*

  The four international project partners are located in York, UK (Dr. Dodds, Dr. Kudenko, Prof. Paige) and Leuven, Belgium (Dr. Mühlberg). The visits have been scheduled so that between those described below, and those described in Section 4.1.2.3, there can be the necessary interaction to carry out all work packages that require collaboration. The Research Assistant will travel to York once per year for consultation with our partners, and once to Leuven in year 3 regarding (WP-8/9A). Dr. White's travel is planned as follows:

  – *York*, to meet with Dr. Kudenko in years 1 and 3 regarding WP-3A and WP-8, respectively.
  – *York*, to meet with Dr. Dodds in year 1 regarding WP-4A.
  – *Leuven*, to meet with Dr. Mühlberg in year 2 regarding WP-8.

  The travel expenses for the 8 European trips will be approx. 1,000 Euro per trip (including accommodation). This totals 8,000 Euro for all travel related to the project's international cooperations.

- *Summer schools*

  The Research Assistant working on the project will apply to a distinguished international summer school related to the topic of Program Analysis and Verification, Machine Learning or Pattern Recognition, for example, the *Machine Learning Summer Schools* (`http://www.mlss.cc/`). The projected costs are 2,000 Euro for this trip, including travel, accommodation and participation fees.

In summary, the requested funds for travel are 29,500 Euro overall.

4.1.2.3 Visiting Researchers. In addition to the funds for visiting the international project partners, we request funds for the following one-week visits of our project partners to Bamberg as follows:

- *Dr. Dodds to Bamberg* in years 1 & 3 regarding WP-4A and WP-9A/B, respectively.
- *Dr. Kudenko to Bamberg* in years 1 & 2 regarding WP-3A/B and WP-7, respectively.
- *Dr. Mühlberg to Bamberg* in years 1 & 3 regarding WP-2A/4A and WP-9A/B/C, respectively.
- *Prof. Paige to Bamberg* in year 3 regarding WP-11.

Analogously to above, these costs are 1,000 Euro per trip, for a total of 7,000 Euro.

4.1.2.6 Project-Related Publication Expenses. 500 Euro p.a. are requested for project related publication expenses to enable us to publish in some of the increasing number of good-quality open-access journals. This totals 1,500 Euro over the application period.

5 Project Requirements

5.1 Employment Status Information

(a) Lüttgen, Gerald, Universitätsprofessor (W3), University of Bamberg (lifelong civil servant)
(b) White, David, Akademischer Rat a. Z., University of Bamberg (fixed-term civil servant, until 30th November 2015)

5.3 Composition of the Project Group

Both authors will be the only people working on the project, who will *not* be paid by the DFG (with the exception of Dr. White for the final 10 months of the project, as detailed in Section 4.1.1).

5.4 Cooperation with Other Researchers

This proposal was inspired in a large part by discussions with our international partners at York in the context of the DFG-funded *initiating international collaboration* project on "Advanced Heap Analysis and Verification" (grant no. LU 1748/2-1).

5.4.1 Researchers Cooperating on this Project

The project team will collaborate with five internationally leading researchers:

Dr. Mike Dodds, University of York, UK. Dr. Dodds is a research lecturer working on concurrency theory, program logics and algorithm verification. His expertise in software verification will enable us to correctly identify how the results of our heap analysis algorithms can inform current and future generations of automated heap verification tools. In particular, he will advise us on case studies suitable for the evaluation of our tool on device drivers in both source code (WP-4A) and object code (WP-9A).

Dr. Daniel Kudenko, University of York, UK. Dr. Kudenko is currently heading the Reinforcement Learning Group at York and has research interests in machine learning, multi-agent systems, user modelling and artificial intelligence for games. His expertise in machine learning will be of great benefit to us when we are designing the machine learning and pattern matching algorithms to cope with the larger class of data structures utilized in real-world software (WP-3A/B). He will also be able to advise us on machine learning algorithms for utilizing payload data (WP-7), and on how to make our algorithms robust against the imprecise information available when dealing with object code (WP-8).

Dr. Jan Tobias Mühlberg, KU Leuven, Belgium. Dr. Mühlberg is a Post Doc in the DistriNet Research Group at Leuven. His research interests include automated software verification, separation logic, program analysis, heap abstraction and analysis of object code programs. As a former PhD student of Prof. Lüttgen, he was responsible for the development of the SOCA (Symbolic Object Code Analysis) Verifier [PSP2, PSP3], which he has employed to verify pointer safety properties in object code of device drivers. He will be our key partner for the work package on transforming the approach to object code level (WP-8) where his experiences with SOCA will be of great benefit to us. He will also be able to provide advise for the work packages targeting device drivers and their evaluation (WP-2A/4A and WP-9A) as well as object-code case studies (WP-9B/C).

Prof. Richard Paige, University of York, UK. Prof. Paige is a Professor of Enterprise Systems at York. His interests lie at the intersection of software engineering and formal methods, where he researches the links between model-driven engineering and search-based software engineering, with an emphases on security. In particular, he is interested in applying these ideas to the runtime-based analysis of complex software systems together with us on WP-11.

Prof. Ute Schmid, University of Bamberg, Germany. Prof. Ute Schmid is Head of the Applied Computer Science/Cognitive Systems group at Bamberg. Her research areas include learning structural data and structural prototypes, planning and applications of machine learning. She will be able to advise us regarding the machine learning and pattern matching algorithms to be developed in WP-3A/B, WP-7 and WP-8.

5.4.2 Past Collaborators

Within the past three years, the applicants have worked with the following national and international researchers: Dr. Antti Siirtola (Aalto University, Finland); Prof. Walter Vogler (University of Augsburg, Germany); Prof. Michael Mendler (University of Bamberg, Germany); Prof. Gianfranco Ciardo (University of California at Riverside, USA); Prof. Willem-Paul de Roever (University of Kiel, Germany); Dr. Jan Tobias Mühlberg (KU Leuven, Belgium); Dr. Milan Ceska (Masaryk University, Brno, Czech Republic); Dr. Mike Dodds, Prof. Richard Paige, Dr. Detlef Plump, Prof. Colin Runciman, Prof. Richard Wilson, Prof. Jim Woodcock (University of York, UK) and Dr. Antti Puhakka (Datactica Ltd, Tampere, Finland)

5.5 Scientific Equipment

All equipment necessary for carrying out the project will be provided by the University of Bamberg.