# Project Description — Follow-Up Project Proposal

Prof. Dr. Gerald Lüttgen, Professor (W3/permanent), University of Bamberg
Thomas Rupprecht, PhD Candidate (E13/temporary), University of Bamberg
Dr. David White, Habilitation Candidate, University of Bamberg

*DSI2: Learning Data Structure Behaviour from Executions of Pointer Programs*

## 1 State of the Art and Preliminary Work

This is a follow-up grant proposal (renewal) to DFG project LU 1748/4-1 of the same title, of which approx. 80% has been completed to date and which will finish at the end of May 2017. The granted current support pays for one doctoral student (Junior Research Assistant, 36 months, Thomas Rupprecht) and one postdoctoral student (Senior Research Assistant, 10 months, Dr. David White). This section reports on what we have achieved so far and reviews some new related work that is relevant for the follow-up research proposed here.

According to the original proposal, 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 behavior, so as to enable a more accurate classification." A common theme in that proposal is the tight coupling between the identification of a data structure and its operations. Fortunately, we have been able to remove this and instead view the two identification aspects as separate processes. The result is a significant increase in our analysis' accuracy and a drastically reduced search space for operation detection.

The research project is on schedule to fully deliver, with the exception of the work package "Utilizing Payload Data" (WP 7), which has been rescheduled into the follow-up project (see WP I/5 in Sec. 2.3 below). There are currently three peer-reviewed publications [PSP2, PSP3, PSP4] associated with the grant, which have all appeared in high-quality international conferences, and one further peer-reviewed poster paper [PSP5] published at the renowned *ACM SIGSAC Conference on Computer and Communications Security* (CCS '16). In addition, four Bachelor's theses have been completed on the topic [5, 14, 35, 42], and one Master's theses is currently in progress. For those work packages with so far unpublished results, we include details of planned publications in the now following description of the project's scientific results.

**Project context.** To manage complexity in software, developers invariably employ the structuring mechanisms offered by data structures. Thus, the understanding of these components plays a critical role in the understanding of a program as a whole. In this project, we are interested in dynamic data structures only and wish to learn two key types of information about the dynamic data structures employed by a program. The first considers only a data structure's shape, e.g., whether there is a recursive pointer establishing the linkage of a linked list or whether two data structures exhibit a relationship such as parent-child nesting. While this information alone can lead to a correct identification, we must often also examine semantic aspects. This forms the second key type of information we wish to learn, e.g., we may distinguish a singly linked list (SLL) that functions as a queue from one that functions as a stack by identifying the operations that manipulate the data structure.

Data structure usage information has several applications. Chief among these are program understanding and debugging, which may be facilitated by, e.g., visualizing and animating the data structures that appear on the heap, inspecting the manipulation caused by a data structure operation, or employing breakpoints set based on data structure properties. We will use the term

*program comprehension* when targeting source code, and *reverse engineering* when targeting stripped binaries, i.e., those with debug information removed. Program comprehension is especially useful when applied to C code, as its analysis can be challenging due to ad-hoc usages of pointers and type casts, as well as placing memory allocation in the hands of the programmer. These aspects are frequently encountered in legacy code that employs non-standard implementations of data structures, and in operating systems and device driver code where efficiency is paramount. For reverse engineering, any hints at what object code does, are highly useful. This is not only true when trying to recover source code when binaries are lost – as happened several times in the computer games industry [`www.cs.vu.nl/~herbertb/projects/rosetta`] –, but even more so in the context of malware as their control structure is often obfuscated.

The analysis output of data structure identification may be leveraged to benefit further types of analyses. In *formal program verification*, for example, the code that manipulates dynamic data structures can complicate the verification process. Thus, by noting that data structure shape is an invariance property, program assertions in the form of pre- and post-conditions can be automatically generated to aid the verification engineer [PSP3]. *Program optimization and profiling* can also benefit from information regarding data structure usage, by delivering, e.g., recommendations for data structure replacements [23] or profiling at the granularity of a specific data structure [34]. Moreover, the field of *signature generation*, e.g., for identifying malware [13], may be enhanced by including data structure information in the analysis.

**Status at project start.** The initial project was proposed from the experience obtained with working with our first research prototype named *Data Structure Operation and Location Identification* (dsOli) [PSP1]. dsOli is a dynamic analysis comprised of an online trace recording phase, which captures relevant program events from an executing program, e.g., memory writes and allocations, and an offline trace analysis phase in which data structures and their associated operations are identified. The analysis works on the intuition that a particular data structure operation is invoked multiple times, and each invocation is similar when viewed under an appropriate abstraction. Thus, data structure operations are repetitive and may be identified using a machine learning technique designed to locate repetition. However, repetitive subsequences of the program's trace due to non-data structure manipulating code will inevitably also be detected. To classify the potential operations and, thus, prune away irrelevant repetition, a template matching scheme is applied, which is based on the concrete modifications to heap structures observed during the potential operation. For example, by matching a template of an SLL with $n$ nodes before the potential operation and then one with $n+1$ nodes after the potential operation, we can conclude that this operation is responsible for inserting one node into an SLL.

While dsOli worked well in practice, it contained a number of shortcomings that we would seek to improve in the project. The most important shortcomings were the reliance on source code, the inability to deal with a greater variety of data structures and nested combinations of data structures, and an insufficient memory abstraction that cannot handle situations in which a node of a data structure does not occupy an entire memory chunk, as occurs when *structs* are embedded or when custom memory allocators are employed. Furthermore, dsOli relied heavily on the correct identification of operation boundaries, i.e., avoiding program states in which a data structure's shape is temporarily degenerated due to a manipulation operation. While the related work [7, 20, 22, 27] already addressed a greater variety of data structures, similar shortcomings were – and are still present today – regarding nested combinations, memory abstraction, and the reliance on avoiding temporally degenerated data structure shapes. However, some of this related work [7, 20, 22] can analyse program binaries without requiring source code.

**Results of work packages.** We now summarise the results of the main work packages of the original proposal. The following discussion is structured in terms of results obtained when considering source code as input and, resp., stripped binaries as input. Tool support for interpreting the analysis output and integration with other analyses intersects these two main topics; so rather than presenting the results chronologically, we delay their presentation.

*Data structure identification for programs with source code.* After studying the requirements for operating system (OS) software, application software and legacy software in "Software Domain Requirements" (WP 2A/B), we decided that the identification of (cyclic) lists, trees, skip lists and arbitrarily deep nested combinations of these should all be in scope for our approach. In addition, we found re-occurring non-classical data structure implementation patterns in C code, which would guide our design decisions. To motivate these, we recall the classic implementation method of a linked list, i.e., where each node of the list resides in a separately allocated chunk of memory and the node occupies the entirety of its associated chunk. The first non-classical implementation concerns generic cyclic doubly linked lists (DLLs) commonly found in OS software, e.g., the Linux kernel list, which may be embedded in other structs to provide a type of high-performance generic programming. The result is that a data structure node might no longer occupy the entirety of its associated memory chunk. The second are highly optimised data structures designed to exploit a specific machine architecture, e.g., combining several nodes into one memory chunk that is cache aligned [11]. Lastly, custom memory allocators result in structures where a memory chunk may contain multiple nodes of multiple different data structures.

These observations led to the first development of "Analysis Technology (Source Code Available)" (WP 3A/B), which is a rich memory abstraction capable of modeling non-standard implementations. By tracking the chains of linkage in a data structure, and in contrast to equating nodes to memory chunks as is done in [20, 22, 27], we allow a memory chunk to contain multiple nodes of potentially multiple data structures simultaneously. These chains essentially correspond to SLLs and form the basic building blocks of data structures in our abstraction, which we term *strands*. With these to hand we may consider the relationships between pairs of strands, i.e., *strand connections*, which are responsible for building more complex data structures. This idea may be exemplified and differentiated from existing approaches [7, 20, 22, 27] by considering a DLL. In these approaches, a DLL simply is a sequence of nodes that collectively have the property DLL, while we consider it as two strands that happen to intersect in a way consistent with a DLL.

One of the key problems faced in data structure identification is that manipulations temporally obscure a data structure's true form, which we term a *degenerate shape*, in contrast to a *stable shape*. For example, a DLL loses its key structural property during insertion and deletion of a node. To counter this, approaches like dsOli [PSP 1], DDT [22] and MemPick [20] only perform identification when it is likely the shape is stable, but this cannot be relied on in general. Thus, as our second contribution in "Analysis Technology (Source Code Available)" (WP 3A/B), we proposed an evidence-based approach to identifying data structures, which is tolerant of the inclusion of degenerate shapes. The idea is to associate evidence with an observation of a data structure based on its structural complexity. For example, a DLL has higher structural complexity than two lists that intersect in some arbitrary way. The evidence is then aggregated within a program time step across data structures that perform the same role, e.g., across the child data structures below a parent. Finally, the evidence is aggregated over time by establishing correspondences between data structures over multiple time steps. All these steps serve to reinforce the evidence for stable shapes and override the evidence for degenerate shapes. Ultimately, we select the interpretation for a data structure with maximum supporting evidence according to a hierarchy, which also enables an elegant natural language naming of the data structure.

We combined our contributions of "Analysis Technology (Source Code Available)" into a tool named *Data structure Investigator* (DSI) [PSP4], which targets data structure identification in C programs. DSI's effectiveness has been proven via "Case Studies and Technology Transfer (Source Code Available)" (WP4 A/B), where it was applied to samples taken from textbooks, synthetically generated samples, samples from the literature, real-world samples such as *bash* [`www.gnu.org/software/bash`], and device-driver like software such as *libusb* [`www.libusb.info`]. For all samples, DSI's memory abstraction is rich enough to model the data structures. Furthermore, we showed that the evidence accumulated by DSI always led to the correct data structure interpretation. On the challenging libusb sample that employs a DLL with two nested child DLLs, the evidence supporting the correct interpretation comprised 88% of the total accumulated evidence.

*Data structure identification for stripped binaries.* The key artifact required by DSI that is not available when working with object code is type information; however, as part of "Analysis Technology (Object Code Available)" (WP 8) we aimed to remedy this by interfacing with a tool that is specialised in type recovery. While there exists a survey on type recovery tools from binaries [8], it does not consider the recovery of structs to the level of detail we require. Therefore, we conducted our own study that we expect to publish at, e.g., the IEEE Intl. Conf. on Software Analysis, Evolution, and Reengineering (SANER). Based on the outcome of our study we selected the type recovery tool Howard [39] by the VUSec research group at VU Amsterdam, due to its ability to recover nested structs which drive our rich heap abstraction. Since Spring 2016 we have been collaborating with VUSec's Prof. Herbert Bos and Mr. Xi Chen on employing the output of Howard as part of a new prototypic front-end for DSI that is capable of handling x86 binaries via Intel's dynamic instrumentation framework PIN [26]. As part of "Case Studies and Technology Transfer (Object Code Available)" (WP 9A/B), we applied our prototype to the samples we employed earlier for evaluation at source code level with highly encouraging results. To highlight the utility of this approach we have recently also applied it to simple malware [PSP5], both with source code (WP 4C) and binary only code (WP 9C), where we aim to aid reverse engineering by, e.g., visualising data structures. Notably, our approach involving binaries also opens up the door of analyzing software written in C++, as is the case for many malware programs.

In the latter work, we found that many data structure implementation techniques cannot be handled by existing type recovery tools. A key problem is that they recover logical types, which often have a many-to-one mapping to the true original types, e.g., variables of the same type but allocated at different sites may end up with different logical types. Thus, a type refinement or merging is frequently performed. As part of our collaboration with VU Amsterdam, we spawned some interesting ideas for improving the type refinement in the current implementation of Howard; for example, we believe that the type recovery may be improved by a machine learning approach (see WP I/2 in Sec. 2.3 below). We have already started developing suitable machine learning technology in the context of a Bachelor's thesis [14] which, in contrast to existing approaches, is able to detect repetition over potentially multiple entry pointers to a data structure. The repetition is indicative of data structure operations and can act as hints for logical type merges. We expect to publish this approach, e.g., at the Intl. Conf. on Machine Learning (ICML).

*Applications of analysis output.* In DSI, and in contrast to dsOli, we may view operation detection as an additional analysis based on DSI's output. Current operation detection relies on the transitions between stable and degenerate shapes which are exhibited by all data structures, with the exception of SLLs that will be handled by carefully selected heuristics. This greatly enhances the accuracy of operation detection by prohibiting operation boundaries from being proposed when degenerate shapes are present. In addition, it automatically handles the case of operations implemented in a recursive style, which was problematic in dsOli. We expect these results to be ready within the next few months and published in the context of a journal article covering the DSI approach and expanding upon [PSP4].

The second half of "Case Studies and Technology Transfer (Source Code Available)" (WP 4A/B) evaluates the usefulness of DSI's analysis output for informing further analyses, with a specific target being formal program verification. To this end and in parallel to the work performed in WP 3, we first investigated the applicability of dsOli for informing the formal verification tool VeriFast [21] developed at KU Leuven, which allows for the automated verification of C programs on the basis of separation logic. This work was done in collaboration with Dr. Mühlberg (KU Leuven, Belgium) and Dr. Dodds (U. York, England). By employing dsOli's output we showed that annotations can be automatically generated, covering pre- and post-conditions of data structure manipulating functions, shape predicates describing the structure of a data structure, and proof machinery to help VeriFast discharge proofs. This led to almost push-button verification for simple data structure programs, and in other cases only small modifications were required from the verification engineer to correct the annotations. Some of the samples were quite complex in nature, including an SLL extracted from the key-value store Redis [`www.redis.io`], for which we

were able to automatically and correctly generate 20 out of the required 22 annotations. The results were reported at the *Intl. Conf. on Software Engineering and Formal Methods* (SEFM 2015) [PSP3]. With DSI now complete, a Bachelor's thesis has shown a proof-of-concept as to how our approach for dsOli can be reworked for DSI [5].

As part of "Tool Support for Analysis Output (Source Code Available)" (WP 5), we provided a tool for interpreting the analysis output of dsOli, which was presented at the *Intl. Conf. on Program Comprehension* (ICPC 2014) [PSP2] and shows how a program's source code can be overlaid with discovered operations along with visualizations of the heap. The user may replay the program and interact with the heap view and source view to aid program comprehension. A similar tool named DSIviz was developed for DSI's output in the context of a Bachelor's thesis [42]. In contrast to the prior tool which uses a generic algorithm for heap layout, DSIviz improved upon the state-of-the-art [2, 27, 31, 38] by employing DSI's rich heap abstraction to guide the layout, and uses animation to show the heap modification to the user. More importantly, closely related approaches in terms of folding heap graphs for memory abstraction [27], consideration of binaries [38] and user interface design [31] all do not handle degenerate shapes and, thus, lack a reliable tracking of data structures across time steps. We expect to submit a conference paper to ICPC 2017 that describes our visualization process. To realise DSIviz we also developed an API that allows DSI's output to be consumed by further software analysis tools.

"Tool Support for Analysis Output (Object Code Available)" (WP 10) turned out to be similar to that required of source code. Thus, we were able to reuse DSIviz and the associated API here. The task of "Aggregating Information over Multiple Traces" (WP 6) is currently being studied in a Master's thesis; the work is greatly simplified as DSI's API allows access to not only the end analysis result, but also to all intermediate abstractions that are mandatory for this work package.

**Summary of contributions.** Our work on DSI has improved upon the state-of-the-art in two key areas. Firstly, we showed that an evidence-based approach to dynamic data structure identification permits the inclusion of degenerate shapes in the analysis, thus providing a more robust analysis than approaches that seek to avoid degenerate shapes [7, 20, 22, 27]. Secondly, DSI's rich memory abstraction which, in contrast to [20, 22, 27], considers data structure nodes as subregions of memory chunks, allows for the identification of many complex data structure implementations frequently arising in C programs. However, our prototypic integration of DSI and Howard [39] with the aim of making DSI capable of handling object code, found many situations in which current type recovery techniques [24, 25, 39] are insufficient. Furthermore, we demonstrated that DSI's output has the potential to aid the formal verification of data structure manipulating code. Last but not least, program comprehension and reverse engineering are eased by the intuitive, data structure driven visualisations and animations of DSIviz. The remaining months on the current project will largely be used by us to author the above-mentioned scientific papers and submit them to international conferences and journals.

**Related work for the proposed follow-up project.** The follow-up project proposed below focuses on broadening and deepening the DSI approach for the domains of reverse engineering, malware analysis and automated verification. This section reviews the most relevant related work that was either insufficiently discussed in the original proposal or that has been published since submitting the original proposal in early 2013: tools on type recovery for binaries, approaches to malware identification and classification, and formal verification techniques for pointer programs.

*Type recovery tools for binaries.* One objective of the proposed follow-up project will be the application of DSI to reverse engineering, i.e., the identification of dynamic data structures in binaries and, especially, in stripped malware binaries. To do so, it is compulsory to recover the type information required by our DSI approach, in particular for basic types and structs [8]. Several tools exist that can help with this, including specialised tools such as Howard [39] and Rewards [25] and more general tools such as DDT [22], MemPick [20] and Artiste [7]. A central task within these tools is aggregating type information, for which MemPick considers type-aware instructions. In contrast, Howard, DDT and Artiste use allocation sites for identifying the above-

mentioned logical types and apply type merging techniques on the basis of employed interface functions, pointer target types and instructions. In practice, problems arise when macros or inlining, nested structs and custom memory allocators are present in the program under analysis. In addition, the above tools typically consider only singly and doubly linked lists and various types of trees, but not skip lists, nestings of lists, or combinations of trees and lists such as in topological sort algorithms. Our DSI approach and the work proposed below aim at lifting these restrictions.

*Malware identification.* Computer and communications security is a key concern in today's digital society, which is threatened by the massively increasing amounts of malware circulated via the Internet. With anti-malware companies receiving tens of thousands of individual malware samples every day, automated clustering techniques are being used to recognise related malware, e.g., polymorphic variants, and to create summary signatures that identify many individual samples from such a family [3]. Malware samples are identified by signatures that capture essential features of the distributed malware file or the program's behaviour, such as infection vectors, system calls or observable network behaviour [10, 32, 33]. Recent research has shown that malware can actively evade behaviour-based detection and even mimic the behaviour of benign software [28], while static analysis [18], data structure information [13], and specifically an analysis of data structure manipulations [37] may lead to more robust clustering and signature generation. Since malware is becoming more complex and employing sophisticated components that also rely on dynamic data structures, malware analysis is an obvious future target for our DSI approach.

*Formal verification of pointer programs.* Handling dynamic data structures presents a major challenge in the formal verification of pointer programs, for which separation logic [30] has emerged as the key way to tackle this challenge. Separation logic tools, e.g., Space Invader [43] and SLAyer [4], typically generate candidate verification conditions by shape analysis [36] and permit the automatic checking of memory safety. To address issues with scalability, over-abstraction and performance, shape analysis tools have added a technique called abduction [9] and transitioned from working on separation logic to using specialised forms of graphs, e.g., the Predator tool [15], and tree automata, e.g., Forester [19]. Recent research extends shape analysis with support for reasoning about payload information [1, 6] and generic inductive shapes [45], using machine learning techniques. However, these automatic approaches are still very much limited when dealing with real code. A better alternative are often automated theorem provers built around separation logic and explicitly supporting C code, such as VeriFast [21]. VeriFast modularly checks via symbolic execution that each function in a C program satisfies its pre- and post-condition, which are given as code annotations in separation logic. As suggested above, DSI's analysis output can be used to auto-generate those verification conditions that relate to data structure shape and memory safety. These are meant to be extended by a verification engineer with the intent to verify properties of the entire program, e.g., functional correctness.

## 1.1 Project-Related Publications

### 1.1.1 Articles Published by Outlets with Scientific Quality Assurance

**[PSP1]** D. H. White and G. Lüttgen. *Identifying dynamic data structures by learning evolving patterns in memory.* In 19th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '13), vol. 7795 of LNCS, pp. 354–369, Springer, 2013. (DOI: 10.1007/978-3-642-36742-7_25. Acceptance rate: 24%)

**[PSP2]** D. H. White. *dsOli: Data structure operation location and identification.* In 22nd Intl. Conf. on Program Comprehension (ICPC '14), pp. 48–52, ACM, 2014. (DOI: 10.1145/2597008.2597800. Acceptance rate: 48%)

**[PSP3]** J. T. Mühlberg, D. H. White, M. Dodds, G. Lüttgen and F. Piessens. *Learning assertions to verify linked-list programs.* In 13th Intl. Conf. on Software Engineering and Formal

Methods (SEFM '15), vol. 9276 of LNCS, pp. 37–52, Springer, 2015. (DOI: 10.1007/978-3-319-22969-0_3. Acceptance rate: 18%)

**[PSP4]** D. H. White, T. Rupprecht, and G. Lüttgen. *DSI: An evidence-based approach to identify dynamic data structures in C programs.* In 25th Intl. Symp. on Software Testing and Analysis (ISSTA '16), pp. 259–269, ACM, 2016. (DOI: 10.1145/2931037.2931071. Acceptance rate: 25%)

**[PSP5]** T. Rupprecht, X. Chen, D. H. White, J. T. Mühlberg, H. Bos and G. Lüttgen. *POSTER: Identifying dynamic data structures in malware.* In 23rd ACM SIGSAC Conf. on Computer and Communications Security (CCS '16), pp. 1772–1774, ACM, 2016. (DOI: 10.1145/2976749.2989041. Acceptance rate: 41%)

#### 1.1.2 – 1.1.3 Other Publications & Patents

**[PSP6]** D. H. White, T. Rupprecht, and G. Lüttgen. *dsOli2: Discovery and comprehension of interconnected lists in C programs.* In 18th Coll. on Programming Languages and Foundations of Programming (Kolloquium Programmiersprachen, KPS '15), 2015. Proceedings available online at `www.complang.tuwien.ac.at/kps2015/proceedings`.

# 2 Objectives and Work Programme

## 2.1 Anticipated Total Duration of the Project

This is a follow-up proposal (renewal) to the DFG project *Learning Data Structure Behaviour from Executions of Pointer Programs*, which has been funded for 36 months until May 2017 (grant no. LU 1748/4-1). Further DFG funding is now sought for extending this project by 36 months.

## 2.2 Objectives

**Aim.** The project's aim is to develop novel techniques and tools to automatically and reliably identify dynamic data structures in a given pointer program, by analysing traces of its execution. The project so far has developed *Data Structure Investigator* (DSI), a sophisticated software tool that has introduced two major novelties: (a) an advanced memory abstraction, so as to deal with the many non-standard data structure implementations in C code, and (b) an evidence-based identification approach, which is tolerant of degenerate shapes that occur when data structure manipulations temporally obscure a structure's true form. DSI has been prototypically applied to three uses cases: program comprehension, for making complex C code legible by visualising its list-based data structures; formal verification, for auto-generating verification conditions related to data structure shape; and reverse engineering, for analysing simple malware components.

The follow-up project, for whose funding we apply here, shall significantly enhance DSI and its underlying approach for the domains of reverse engineering and formal verification, and make our tool accessible to experts in these domains. This overall aim involves three goals:

I. The DSI approach shall be extended to cover richer data structures such as arrays, sorted lists/trees, balanced trees and complex nestings of structures. It shall also be made more robust and flexible by utilising memory snapshots, i.e., partial rather than complete program execution traces. This will allow the investigation of more complex software and data structures in practice, including software involving input-dependent execution branches. We also want to deepen our approach to work with complex combinations of data structures, e.g., topological sort trees and hash maps.

II. DSI shall assist security engineers in reverse engineering malware. There is an urgent need to understand the numerous malware that threaten computer security and which are increasing steeply in complexity. Doing so requires advances in inferring type information from binaries and interfacing DSI to the widely used IDA Pro tool for the reverse engineering and debugging of binaries. We expect that the identification of data structures in malware will also help us to advance techniques for malware signaturing and classification.

III. DSI shall alleviate verification engineers from having to manually provide verification conditions related to data structure shape and behaviour. A challenge will be to support the many coding styles found in pointer programs. Interfacing DSI to the popular VeriFast verifier will allow its users to speed up formal verification for safety- and security-critical software.

Our deepened approach and enhanced DSI tool shall be evaluated by means of case studies involving system-level software and malware. Alongside, the potential for technology transfer shall be explored by porting DSI to the commercially popular Windows and Android platforms.

**Objectives.** The concrete objectives of the follow-up project are as follows:

*1. Enhancing DSI's robustness (WPs I/3 & WP I/6).* So far, DSI analyses single traces only and solely those that capture every single pointer-write or memory (de)allocation event in a program. This fine granularity is sometimes unnecessary, resulting in an analysis speed-up when being able to skip events, or impractical when a program must be run several times with different inputs so that its various control paths are covered. In addition, operations on more complex, nested data structures may be compound, and the current DSI tool may not correctly identify compound operations in their entirety. The first objective is thus to design more robust analysis support, which takes partial and multiple traces as well as compound operations into account.

*2. Broadening DSI's scope (WPs I/4 & WP I/5).* The current DSI approach ignores complex nestings of data structures, such as linked lists going through tree nodes, as used, e.g., in topological sort algorithms. DSI does also not handle arrays that are employed together with pointers to implement, e.g., hash maps, or other complex data structures, e.g., balanced trees. In addition, payload information is currently not inspected by DSI, i.e., semantic information relevant to data structures such as sorted lists or sorted trees is not utilised. We will develop improved analysis algorithms that will be sensitive to these structural and semantic aspects.

*3. Reverse engineering (WPs I/2, II/1 & II/4).* The project to date has prototypically demonstrated how DSI could be interfaced to a state-of-the-art type inference tool, Howard, and Intel's PIN instrumentation framework, so as to analyse dynamic data structures in binaries. Our studies have observed, however, that Howard is frequently unable to correctly discover types, especially nested structs, and thus impedes DSI's memory abstraction. The third objective will explore techniques to overcome this limitation and tightly integrate our DSI tool for binaries with the IDA Pro reverse engineering tool, which is the *de facto* industry standard but handles pointer structures poorly. We also plan to port DSI to the Windows and Android platforms, so as to enhance its applicability and initiate technology transfers.

*4. Analysing & classifying malware (WPs II/2 & II/3).* The DSI plug-in for IDA Pro will be the basis for analysing malware. Modern malware consists of components, some of which implement data storage and manipulation facilities using pointer structures. On the basis of extensive case studies, we wish to explore how successful our DSI plug-in can be used for malware analysis and what DSI's limitations are. Malware samples shall be taken from publicly accessible repositories that also contain their source code, so that a ground truth is given. In addition, this fourth objective shall investigate the utility of DSI's analysis output for helping to construct novel signatures of malware components and for classifying malware.

*5. Automated verification of pointer programs (WPs III/1-3).* The project so far has prototyped a tool for auto-generating verification conditions from pointer programs written in C, including pre- and post-conditions of C functions, for use by the state-of-the-art verifier VeriFast. Results show

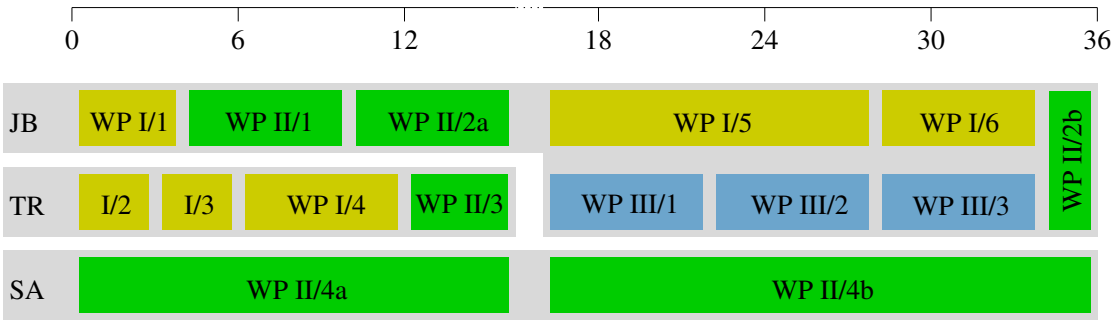


Figure 1: *Proposed scheduling of work packages. The work packages will be conducted by Thomas Rupprecht (TR) and Jan Boockmann (JB), who are the PhD student and student assistant currently working on the project, resp., as well as a Student Assistant (SA). Mr. Boockmann will only be working 50% part-time for the first 16 months, while finishing his Master's studies, and will always conduct two work packages in parallel between months 17 and 34.*

that shape and behavioural properties of data structures can in principle be reliably synthesised, but obstacles arise due to the many coding styles in real software. This fifth objective shall overcome the current limitations of our generator by developing a theory of strands that formalises our memory abstraction. This will address the variety of coding styles and expand our technique from non-nested list structures to nested dynamic data structures including trees. Our approach shall be compared carefully to recent advances in shape analysis.

*6. Tool engineering & evaluation (WPs I/1-6, II/1, II/2, II/4, III/2 & III/3).* Mature, user-friendly tool support is essential for DSI, if it is to be taken up by security and software engineers. Hence, the last but not least objective is to significantly enhance DSI's tool support. This involves the development of a GUI for traversing sequences of memory graphs and zooming into and out of graph regions, and extending the existing layouting techniques to cope with the more advanced data structures to be studied in this follow-up project. In addition, DSI shall be interfaced to widely used reverse engineering and formal verification tools, as mentioned above. This will allow us to explore several case studies in these domains and also in malware analysis, so as to systematically evaluate the strengths and weaknesses of the DSI approach.

**Outcome & Impact.** The project's outcome to date are advanced algorithms to automatically identify data structures and their associated operations from traces of automatically instrumented executions of pointer programs [PSP1, PSP4], as well as prototypic support for the three use cases described above: program comprehension [PSP2], formal verification [PSP3] and malware analysis [PSP5]. The developed technology is implemented in the novel software tool *Data Structure Investigator* (DSI), which has been successfully employed to conduct proof-of-concept case studies for all these use cases and currently comprises about 25k lines of code.

The proposed follow-up project will make DSI available to security and software engineers. The core DSI approach will be fundamentally extended to cover richer data structures, matured in its robustness to handle multiple and partial traces, and enhanced wrt. tool support and interfacing to other software analysis tools. This will especially benefit engineers performing reverse engineering in the security domain. We expect that knowledge of data structures will be of great help here and will also allow the advancement of techniques for signatures and malware classification. Extending DSI to binary analysis and interfacing it to the widely deployed IDA Pro tool for reverse engineering will significantly support these goals. Providing high-quality information on shape, operational behaviour and coding style of a pointer program's data structures will also impact software engineers who verify safety- and security-critical software. By automatically generating verification conditions related to the shape and behaviour of a dynamic data structure, for the popular VeriFast verification tool, engineers will be alleviated from labor-intensive and error-prone tasks and thus be able to concentrate on the actual verification job.

## 2.3 Work Programme incl. Proposed Research Methods

The proposed programme of work involves 13 work packages, which are scheduled as depicted in Fig. 1 and structured along three axes, each of which follows one of the goals set out above: Axis I *(yellow)* involves the packages related to the deepening of our DSI approach, Axis II *(green)* focuses on the packages for reverse engineering and malware analysis, and Axis III *(blue)* sums up the packages for pointer program verification. We envisage that our international collaborators participate as follows: Prof. Bos with his expertise in security and type inference from binaries in WPs I/2 & II/1-3, Dr. Mühlberg with his knowledge of malware and formal verification in WPs II/1-4 & III/1-3, Mr. Rafique who specialises in malware analysis and classification in WP II/3, Dr. Rawat with his expertise on dynamic security program analysis in WPs II/2-3, and Dr. White with his insights into the DSI approach in WPs I/1-6.

**WP I/1: Enhanced memory visualisation.** Our data structure visualizer DSIviz presents the programmer with a legible graph layout of the data structures under investigation over their lifetimes; however, it is currently limited to standard linked-list structures. In this work package, we wish to develop more advanced layouting algorithms that also cover, e.g., various forms of trees and skip lists, while tracking a data structure even when stepping through the degenerate shapes of manipulation operations.

In addition, the current DSIviz focuses on layouting only and not also on the user experience. For example, points-to graphs are quickly becoming very large for real programs, and the sheer sizes make their layouting illegible. Our memory abstraction in terms of strands, strand connections and the resulting strand graphs can help us to cope with this complexity, e.g., by naturally folding parts of points-to graphs into strand graphs, unfolding strand graphs into points-to graphs, and zooming into and out of regions of such graphs. Overall, the user will be able to easily focus on the areas of interest and benefit from the continuity of the displayed data structures.

**WP I/2: Type inference from binaries.** This work package shall overcome the limitations of the type recovery tool Howard and similar tools, as explained in Sec. 1, which practically implies for us to develop new type inference methods that directly address the needs of DSI. This is also necessary for porting DSI to Windows (see WP II/4a below) as Howard is only available for Linux. Basic type information, such as the sizes of allocated memory chunks and the detection of pointers, will be extracted from the instrumentation of binary code and enriched by exploiting type-revealing instructions and type sinks.

The discovery of nested structs will, however, require more advanced technology. For this we envisage a novel approach that, as a first step, creates various hypotheses, e.g., several possibilities of how a nested struct aligns inside its outer struct. These will be checked for plausibility by propagating the information via the linkages between memory chunks involving these structs. In a second step, each hypothesis is validated by executing DSI and using the collective results as a fitness measure: the hypothesis with the best interpretation, e.g., in terms of the largest and structurally most complex data structure detected, will then be identified as the most likely type interpretation. This approach will solve both the type merging problem and the detection of nested structs, which is crucial for DSI's ability to detect highly customised data structures such as the Linux kernel list. We believe that our approach will also make a notable contribution to the type inference problem from binaries in general.

**WP I/3: Memory snapshot support.** DSI operates by processing each data structure manipulating event contiguously, which is an incremental task that cannot easily be parallelised. This work package shall develop a technique that is able to correlate non-contiguous time steps, in the sense of aggregating information from different memory snapshots rather than simply comparing information as is done, e.g., in [41]. This will also allow for the correlation of traces from different executions, so as to cover a program's behaviour more completely. Aggregating such multiple analysis results will produce more evidence and thereby enable a higher quality and more robust analysis as well as a more comprehensive understanding of the program under study.

Aggregating memory snapshot information will be performed on DSI's memory abstraction, i.e., the strand graph, and adapt standard maximum common subgraph algorithms [16] to identify the common elements across several program time steps, thus allowing their associated evidence to be summed. Most challenging will be aggregating information across multiple traces because, e.g., entry points to a data structure will no longer have a single unique address. To tackle this problem, canonicalization techniques from the literature will be applied [40]. We will evaluate our enhanced approach by re-considering situations that are difficult for the current version of DSI.

**WP I/4: Rich structures discovery.** This work package will address complex nesting situations that occur in practice, e.g., with the topological sort routine *tsort* from GNU coreutils [`www.gnu.org/s/coreutils`], where an SLL runs through nodes organised as a tree. While DSI is already capable of correctly recognizing strand graphs associated with a given data structure, the final interpretation is sometimes challenging; for example for *tsort*, one needs to resolve the nesting direction between the tree and the list, which should be reported as 'from the tree to the list'. Possible ways to discover directionality are, e.g., detecting the longest running entry pointer to determine the dominant data structure, or tracking access patterns inside data structures. The latter could also benefit deeper data structure inspections such as hot-spot analysis.

To further enrich the capabilities of DSI, we will add the *array* program construct to our strand abstraction. This will enable the detection of frequently used data structures such as hash maps, e.g., in dix/resource.c from the X.org server [`www.x.org`], or n-ary trees, e.g., in massif/ms_main.c from the Valgrind instrumentation framework [`valgrind.org`]. Finally, we wish to inspect additional structural characteristics of data structures such as the balancedness of trees. While all these enhancements are highly worthwhile on their own, we also believe that they will directly enrich the creation of signatures for malware to be tackled in WP II/3.

**WP I/5: Payload support.** The purpose of this work package is to make DSI aware of payloads in dynamic data structures, thus permitting the reasoning about semantic aspects of data structures, e.g., the sortedness of a list or a tree. The challenge lies in the many different forms that payload elements may take; for example, the sorting aspect may concern payloads ranging from primitive data types, to strings, to compound data types. In fact, the payload might not even reside directly in the linkage structure, but instead it might be part of an enclosing struct or accessed indirectly via pointers. Additionally, in the case of compound data types, complex logic may be applied, e.g., for the comparison operation used for sorting.

To enable payload analysis, we will widen our instrumentation frameworks for source and binary code to record non-pointer writes. Initially, we plan to add simple heuristics to detect basic cases of sorting. Later on, this will be enhanced by carefully observing code sections before insert operations, in order to determine which data is consulted and which logical operations are applied. Examples for sorted data structures are found, e.g., in the Carberp and Rovnix malware as described in WP II/2a(+b). Information on payload data will also be added to our data structure visualiser DSIviz (cf. WP I/1).

**WP I/6: Compound operation detection.** While DSI is capable of identifying operations by observing transitions between stable and degenerate shapes, this only reveals atomic data structure operations. However, when dealing with compound data structures, e.g., a list of lists, one global insert operation might consist of an insert for the parent and one for the child. Another example is an operation that involves two separate data structures, such as moving one element between two lists. Possible solutions are the correlation of degenerate shapes between different data structures: as soon as two data structures are simultaneously in a degenerate shape, this will be a hint for a compound operation. Another approach is to record a feature trace that covers data structure traversals and transitions between stable and degenerate shapes, which can then be used as input to the machine learning technique already employed by us in [PSP1] to find repetitive behaviour corresponding to re-occurring operations. This will also enable the setting of breakpoints at more suitable program locations and to generate verification conditions more accurately, as is needed in WPs II/1 and III/2, resp.

**WP II/1: DSI plug-in for IDA Pro.** IDA Pro is *the* industry-standard, commercial software tool for the reverse engineering of binaries and available for multiple platforms including Linux and Windows. It combines an interactive disassembler with a local and remote debugger and a plug-in environment for extensions. One significant shortcoming of IDA Pro is that it naturally works at a very low level, e.g., while it can detect pointers, it is not able to recognise and name dynamic data structures. This is where DSI comes into play: it can take the debugging trace of IDA Pro to reveal dynamic data structures and visualise them to the user; indeed, it could animate the visualisation in synchrony with the instructions executed in IDA Pro's debugger. In addition, DSI can detect the operations manipulating such data structures and relate them to regions in the binary code, which can then be highlighted in the debugger and used for automatically setting breakpoints/watchpoints. The highlighting also applies to typing information that is excavated by DSI (cf. WP I/2). This motivates our desire for a DSI plug-in for IDA Pro and, given the popularity of IDA Pro, will greatly help in transferring our research results into engineering practice.

**WP II/2a(+b): Malware comprehension case studies (revisited).** This work package will employ the DSI plug-in for IDA Pro to analyse complex malware, e.g., Carberp and MyDoom [`github.com/ytisf/theZoo`] and Agobot [`en.wikipedia.org/wiki/Agobot`]. After applying our tool chain to the binaries of these samples, we will inspect their available source code in order to compare DSI's output with our findings. We will also stress test our approach by making the example binaries successively more difficult to analyse, e.g., by increasing the level of compiler optimisation and by applying code and data structure obfuscations [12, 44]. Additionally, we will investigate unknown malware samples for Linux, which will be provided by our project partners.

These case studies will be revisited after having implemented the extended DSI capabilities of WPs I/4-6 and supplemented by further samples, with the expectation of revealing additional data structure information. Candidate examples are a sorted cyclic DLL implementation in the graphics package gfx taken from Carberp, and a sorted SLL implementation in the SNMP (Simple Network Management Protocol) module of the Rovnix malware [`github.com/ytisf/theZoo`].

**WP II/3: Enhanced malware signatures.** Here, we will investigate the utility of DSI's analysis output for enhancing signatures of software components, to help with the classification and identification of malware. Our preliminary results show that modern malware is component-based – using, e.g., an IRC bot or a VNC server as a component – and does indeed employ dynamic data structures [PSP5]. In our observation, various malware components may be identified by the data structures they employ, despite the program obfuscations that are applied when creating malware, and this fact should be utilized when reverse engineering and identifying malware.

We therefore conjecture that applying DSI to compiled malware samples can provide features for robust malware classification and signature generation, even for stealthy samples that may not exhibit distinguishable system calls or network activity when being inspected using tools like IDA Pro (cf. WP II/1). Together with our international partners, we will further investigate application scenarios for emerging threats to mobile computing [17], where DSI-like technology could be employed to screen apps before those are being promoted on third-party markets.

**WP II/4a+b: DSI port for Windows + Android.** Because malware is frequently written for the Windows platform and a promising use case for DSI, we will port DSI from Linux to Windows. More precisely, we will port DSI's PIN-based binary front-end including the type inference module, so as to be able to create the trace of the program under study that is then analysed by DSI's back-end. Note that both PIN and IDA Pro are available for Windows. The porting of DSI will require some engineering effort due to the rather complex libraries and interfaces found in Windows.

Another popular computing platform is Android, which is of interest to us due to the same reasons as Windows. Porting DSI to Android will involve investigating Intel's PIN for Android or an alternative instrumentation framework. Since Android software is not only available in compiled form but – at least at install-time – also in Android's Dalvik/DEX bytecode [`source.android.com/devices/tech/dalvik/dalvik-bytecode.html`], the type information required by DSI can be directly inferred. This interplay between compiled and rich intermediate code makes Android an

interesting target for evaluating the strength and weaknesses of DSI's technology, and in particular its type inference technology. The obfuscation of compiled Android code [gdeveloper.android.com/studio/build/shrink-code.html] will be a further challenge to DSI.

**WP III/1: Modular theory of strands.** This work package will formalise DSI's underlying heap representation and evidence counting mechanism, primarily to significantly enhance DSI's utility for formal program verification. Recall that we have successfully conducted a proof-of-concept for the use of DSI's analysis output to automatically generate and inject pre- and postconditions as well as simple loop invariants into the functions of a given C program (cf. [PSP3] and [5]). These verification annotations can often be discharged by the VeriFast verifier, which saves the engineer much manual effort when proving memory safety properties. However, our prototype has notable shortcomings, including that it only covers simple lists, does not support nesting, and does not cover the many coding styles employed in real C code.

We will address these shortcomings by rethinking our approach mathematically. The idea is to make use of our memory abstraction in terms of strands and strand connections, where the latter describe relationships between strands such as nesting. This abstraction can be formalised as an algebra, with strand connections in essence being operators on strands. We envisage to utilise this modular algebraic structure for auto-generating verification conditions systematically and locally, starting from basic, simple verification templates for strands. Thereby, we avoid the need to construct a specific, complex verification template globally for each (kind of) strand graph and, thus, for each specific coding style, as we do now [5]. The algebraic approach will also enable us to properly pass around contextual information that is required for dealing, e.g., with nesting. DSI will then be able to address implementations of the above-mentioned topological sort algorithm. As an additional benefit, we expect that the algebraic approach will allow us to set our evidence counting, which is at the heart of DSI, on a sound mathematical footing.

**WP III/2: Verification condition generator.** We will use the formalisation of WP III/1 to integrate DSI with Leuven's program verifier VeriFast. Building upon ongoing work to further automate VeriFast [29], we believe that our algebraic approach to strand graphs will lead to a rigorous way of constructing and automatically discharging verification conditions for programs that employ generic as well as customised dynamic data structures. Our assertion generator will also implement support for the semantic data structure features considered in WP I/5, so as to generate the verification conditions covering, e.g., sorting properties of lists and trees, too.

In this context, we will additionally implement some features relevant to practical formal verification, which are already supported by our DSI approach but not yet by the DSI tool. This concerns, e.g., the identification of entry pointers to a data structure in the presence of multiple indirections, so that we can connect program variables to the correct entry points. Our verification condition generator will be evaluated by means of our benchmark of textbook and synthetic C samples considered in [PSP3] and WPs I/4-6, and exercised to verifying real C programs in WP III/3.

**WP III/3: Formal verification case studies.** The aim of this work package is to evaluate in cooperation with KU Leuven our modular approach to program assertions and their generation, as developed in the previous two work packages. This will be done by conducting case studies involving real-world software projects, so as to see whether our tool chain can cope with the large variety of coding styles employed in practice. Our focus shall be on small and/or embedded software for which extensive documentation and test cases are available. Candidate software includes embedded kernels, such as PikeOS [www.sysgo.com/products/pikeos-hypervisor], Contiki [www.contiki-os.org], TinyOS [webs.cs.berkeley.edu/tos]; the Antelope embedded DBMS inside the Contiki project; a small implementation of the Transport Layer Security (TLS) protocol, such as Amazon S2N [github.com/awslabs/s2n]; the Redis data structure store [www.redis.io], which we partially addressed in preliminary work [PSP3]. The idea is to fully verify a number of compilation units and interfaces of the above projects and to evaluate the impact of the results of WP III/2 on the VeriFast verification effort. We conjecture a substantial decrease in manual verification time for software components that heavily rely on dynamic data structures.

This work package will, together with WPs II/1 and II/4, enable the transfer of the DSI technology into engineering practice, in the domains of computer security, reverse engineering and formal verification. Our international collaborators at Amsterdam and Leuven, who frequently cooperate with industry, will help us in assessing this potential.

### 2.4 – 2.6 Not Applicable

2.4 Data handling; 2.5 Other information; 2.6 Descriptions of proposed investigations involving experiments on humans, human materials or animals.

### 2.7 Information on Scientific and Financial Involvement of International Cooperation Partners

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

## 3 Bibliography


[1] P. Abdulla, L. Holík, B. Jonsson, O. Lengál, C. Q. Trinh, and T. Vojnar. Verification of heap manipulating programs with ordered data by extended forest automata. *Acta Informatica*, 53(4):357–385, 2016.

[2] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer. Heapviz: Interactive heap visualization for program understanding and debugging. In *SOFTVIS '10*, pp. 53–62. ACM, 2010.

[3] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS '09*. The Internet Society, 2009.

[4] J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory safety for systems-level code. In *CAV '11*, vol. 6806 of *LNCS*, pp. 178–183. Springer, 2011.

[5] J. Boockmann. Automatic generation of data structure annotations for pointer program verification. Bachelor's thesis, U. Bamberg, Germany, October 2016.

[6] M. Brockschmidt, Y. Chen, B. Cook, P. Kohli, S. Krishna, D. Tarlow, and H. Zhu. Learning to verify the heap. Techn. Rep. MSR-TR-2016-17, Microsoft Research, 2016.

[7] J. Caballero, G. Grieco, M. Marron, Z. Lin, and D. Urbina. Artiste: Automatic generation of hybrid data structure signatures from binary code executions. Techn. Rep. TR-IMDEA-SW-2012-001, IMDEA Software Inst., Spain, 2012.

[8] J. Caballero and Z. Lin. Type inference on executables. *ACM Computing Surveys*, 48(4):1–65, 2016.

[9] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *SIGPLAN Notices*, 44(1):289–300, 2009.

[10] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. A quantitative study of accuracy in system call-based malware detection. In *ISSTA '12*, pp. 122–132. ACM, 2012.

[11] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Making pointer-based data structures cache conscious. *IEEE Computer*, 33(12):67–74, 2000.

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

[13] A. Cozzie, F. Stratton, H. Xue, and S. King. Digging for data structures. In *OSDI '08*, pp. 255–266. USENIX Association, 2008.

[14] L. Dietz. Multidimensional repetitive pattern discovery for locating data structure operations. Bachelor's thesis, U. Bamberg, Germany, April 2015.

[15] K. Dudka, P. Peringer, and T. Vojnar. Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In *CAV '11*, vol. 6806 of *LNCS*, pp. 372–378. Springer, 2011.

[16] H.-C. Ehrlich and M. Rarey. Maximum common subgraph isomorphism algorithms and their applications in molecular science: A review. *Comput. Molecular Science*, 1(1):68–79, 2011.

[17] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan. Android security: A survey of issues, malware penetration, and defenses. *IEEE Communications Surveys & Tutorials*, 17(2):998–1022, 2015.

[18] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In *FSE '14*, pp. 576–587. ACM, 2014.

[19] P. Habermehl, L. Holík, A. Rogalewicz, J. Simácek, and T. Vojnar. Forest automata for verification of heap manipulation. *Formal Methods in System Design*, 41(1):83–106, 2012.

[20] I. Haller, A. Slowinska, and H. Bos. Scalable data structure detection and classification for C/C++ binaries. *Empirical Software Engineering*, 21(3):778–810, 2016.

[21] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, Willem W. Penninckx, and F. Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In *NFM '11*, vol. 6617 of *LNCS*, pp. 41–55. Springer, 2011.

[22] C. Jung and N. Clark. DDT: Design and evaluation of a dynamic program analysis for optimizing data structure usage. In *MICRO '09*, pp. 56–66. ACM, 2009.

[23] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: Effective selection of data structures. *SIGPLAN Notices*, 46(6):86–97, 2011.

[24] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled reverse engineering of types in binary programs. In *NDSS '11*. The Internet Society, 2011.

[25] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *NDSS '10*. The Internet Society, 2010.

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. PIN: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Notices*, 40(6):190–200, 2005.

[27] M. Marron, C. Sanchez, Z. Su, and M. Fähndrich. Abstracting runtime heaps for program understanding. *IEEE Transactions on Software Engineering*, 39(6):774–786, 2013.

[28] J. Ming, Z. Xin, P. Lan, D. Wu, P. Liu, and B. Mao. Replacement attacks: Automatically impeding behavior-based malware specifications. In *ACNS '15*, vol. 9092 of *LNCS*, pp. 497–517. Springer, 2015.

[29] N. Mohsen and B. Jacobs. One step towards automatic inference of formal specifications using automated VeriFast. In *FMICS '16*, vol. 9933 of *LNCS*, pp. 56–64. Springer, 2016.

[30] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL '01*, vol. 2142 of *LNCS*, pp. 1–19. Springer, 2001.

[31] J. Ou. HeapVision: Debugging by interactive heap navigation. Master's thesis, ETH Zürich, Switzerland, March 2014.

[32] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *NSDI '10*, pp. 26–26. USENIX, 2010.

[33] G. Portokalidis, A. Slowinska, and H. Bos. Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *EuroSys '06*, pp. 15–27. ACM, 2006.

[34] E. Raman and D. I. August. Recursive data structure profiling. In *MSP '05*, pp. 5–14. ACM, 2005.

[35] J. Molina Ramirez. Evaluation and development of program obfuscation techniques to break repetitive program behaviour. Bachelor's thesis, U. Bamberg, Germany, July 2014.

[36] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.

[37] A. F. Shosha, C. C. Liu, P. Gladyshev, and M. Matten. Evasion-resistant malware signature based on profiling kernel data structure objects. In *CRiSIS '12*, pp. 1–8. IEEE, 2012.

[38] V. Singh, R. Gupta, and I. Neamtiu. MG++: Memory graphs for analyzing dynamic data structures. In *SANER '15*, pp. 291–300. IEEE, 2015.

[39] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS '11*. The Internet Society, 2011.

[40] W. N. Sumner and X. Zhang. Memory indexing: Canonicalizing addresses across executions. In *FSE '10*, pp. 217–226. ACM, 2010.

[41] D. Urbina, Y. Gu, J. Caballero, and Z. Lin. Sigpath: A memory graph based approach for program data introspection and modification. In *ESORICS '14*, vol. 8713 of *LNCS*, pp. 237–256. Springer, 2014.

[42] K. Welzel. Heap visualisation using interactive memory graphs. Bachelor's thesis, U. Bamberg, Germany, March 2016.

[43] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *CAV '08*, vol. 5123 of *LNCS*, pp. 385–398. Springer, 2008.

[44] L. Zhiqiang, R. Ryan, and X. Dongyan. Polymorphing software by randomizing data structure layout. In *DIMVA '09*, vol. 5587 of *LNCS*, pp. 107–126. Springer, 2009.

[45] H. Zhu, G. Petri, and S. Jagannathan. Automatically learning shape specifications. In *PLDI '16*, pp. 491–507. ACM, 2016.

# 4 Requested Modules/Funds

Funds are requested for the full application period of 36 months.

## 4.1 Basic Module

### 4.1.1 Funding for Staff

***1 Senior Research Associate, Senior RA*** (Postdoc) for 16 months full-time (Postdoktorand/in und Vergleichbare, 100% der regelmäßigen Arbeitszeit)

Thomas Rupprecht, who is the RA on the current project, will continue on the project for 16 months, if it should be funded. He is expected to complete his PhD by May 2017 and will thus be a post-doctoral researcher by the start of the follow-up project. In addition to being fully familiar with the DSI approach and being the developer of the DSI prototype for binaries, he has co-supervised the Bachelor's thesis of the selected Junior RA Jan Boockmann (see below). Mr. Rupprecht will assist Mr. Boockmann in becoming acquainted with the details of the DSI approach and the DSI tool's approx. 25k lines of source code.

***1 Research Assistant*** for 16 months at 50% part-time (sonstige/r wiss. Mitarbeiter/in, 50% der regelmäßigen Arbeitszeit) ***and, subsequently, 1 Junior Research Associate, Junior RA*** (PhD student) for 20 months full-time (Doktorand/in und Vergleichbare, 100% der regelm. Arbeitszeit)

This Junior RA post will be filled by Jan Boockmann, a brilliant current student at Bamberg who is expected to complete his Master's studies in September 2018. He will work as a 50% part-time Research Assistant on the project during his Master's studies, and then continue for the remaining 20 project months as full-time Junior RA. Mr. Boockmann is already employed as the Student Assistant on the project and, for his Bachelor's thesis, he has developed a simple prototype of the verification condition generator that is to be fully developed in work package WP III/2. He is therefore the ideal Junior RA for the follow-up project.

***2 Student Assistants, SAs*** (Studentische Hilfskräfte), each for 44 hours per month for 36 months at the standard rate of 12.16 Euro per hour, for a total of 38,522.88 Euro.

The first SA will assist the RAs with tool development, mainly with the programming and evaluation tasks as well as the case studies of WPs I/1–6, II/1-3 & III/2+3. The second SA will port the DSI tool to the Windows and Android platforms (WP II/4a+b). The SAs will likely be recruited among the students who take the applicant's (Prof. Lüttgen's) modules on software engineering, software analysis and software verification at Bamberg.

### 4.1.2 Direct Project Costs

**4.1.2.1 Equipment up to 10,000 Euro, Software and Consumables.** All staff on the proposed project will be equipped by the University of Bamberg with state-of-the-art PCs and/or laptops, including headsets and webcams needed to communicate with our international collaborators in Leuven, Belgium and Amsterdam, The Netherlands. All software necessary for carrying out the project is available free of charge, with the exception of the commercial software IDA Pro by Hex-Rays SA, which is an industry-standard multi-processor disassembler and debugger. To carry out WPs II/1, II/2 and II/4a, two IDA Pro computer licences (Windows+Linux) are needed, one together with an x64 decompiler license (Linux) and the second with an x64 decompiler license (Windows). These currently cost 2x 1,535 Euro + 2x 2,136 Euro = 7,342 Euro. Given that Hex-Rays increased prices by 18% from 2015 to 2016 and expecting a similar increase for 2017, the total projected cost is 8,664 Euro. No funds for consumables are requested.

**4.1.2.2 Travel Expenses.** Funds are requested to cover the cost of travel that is required for carrying out the project and presenting its results to the international scientific community:

***Conferences, symposia & workshops.*** The project team will actively participate in international conferences, symposia and workshops: the Senior RA in two European meetings and one

overseas meeting in the 16 months during which he will be working on the follow-up project; the Junior RA in three European meetings and one overseas meeting over the full project period of 36 months; and the applicant in two European meetings and one overseas meeting over the full project period. 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, 16,500 Euro is requested in total for all conference, symposia and workshop travel.

Examples of targeted international conferences, symposia and workshops are, in alphabetical order: *Computer Aided Verification* (CAV), *Computer and Communications Security* (CCS), *European Symposium on Security and Privacy* (Euro S&P), *Foundations of Software Engineering* (FSE), *International Conference on Program Comprehension* (ICPC), *International Symposium on Software Testing and Analysis* (ISSTA), *Network and Distributed System Security* (NDSS), *Software Analysis, Evolution, and Reengineering* (SANER), *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS).

***International cooperations.*** The four international project partners are located in Leuven, Belgium (Dr. Jan Tobias Mühlberg and Zubair Rafique) and Amsterdam, The Netherlands (Prof. Herbert Bos and Dr. Sanjay Rawat). For the project duration, five one-week visits to these partners are planned: one visit each to Leuven and Amsterdam by the applicant, one visit by the Senior RA to Leuven, and one visit each to Leuven and Amsterdam by the Junior RA. The travel expenses for each trip will be approx. 1,000 Euro, for an overall cost of 5,000 Euro.

***Summer school.*** The Junior RA will apply to a distinguished international summer school on one or more of the topics Automated Verification, Reverse Engineering and Computer Security. The projected costs are 2,000 Euro for travel, accommodation and participation fees.

In summary, the requested funds for travel total 23,500 Euro overall.

**4.1.2.3 Visiting Researchers.** In addition to the funds for visiting the four international project partners at Leuven and Amsterdam as explained under "*International cooperations*" above, we request funds for one one-week visit of each of these partners to Bamberg. Analogously to above, these costs total 4,000 Euro. Dr. White, who has been the Senior RA on the project until August 2016 and now lives in Taiwan, will be available for consultations in Bamberg once for two weeks in each of the three project years. The projected cost for his travel and accommodation are 3,000 Euro per trip. Hence, the requested funds for visiting researchers total 13,000 Euro overall.

**4.1.2.4 & 4.1.2.5 Not Applicable.** 4.1.2.4 Expenses for laboratory animals; 4.1.2.5 Other costs.

**4.1.2.6 Project-Related Publication Expenses.** 500 Euro p.a. are requested for publication expenses, totalling 1,500 Euro over the application period. This fund will enable us to publish in some of the increasing number of high-quality open-access journals and conference/workshop proceedings that impose moderate charges, such as those appearing in the *Leibniz International Proceedings in Informatics* (LIPIcs) series.

#### 4.1.3 Instrumentation: *Not applicable*

## 4.2 – 4.7 Not Applicable

4.2 Module temporary position for funding; 4.3 Module replacement funding; 4.4 Module temporary clinician substitute; 4.5 Module Mercator fellows; 4.6 Module workshop funding; 4.7 Module public relations funding.

# 5 Project Requirements

## 5.1 Employment Status Information

Lüttgen, Gerald, Universitätsprofessor (W3), University of Bamberg (lifelong civil servant)

## 5.2 First-Time Proposal Data: *Not applicable*

## 5.3 Composition of the Project Group

The applicant will be the only person working on the proposed project, who will *not* be paid out of the DFG's funds.

## 5.4 Cooperation with Other Researchers

### 5.4.1 Researchers Cooperating on this Project

**Past & present cooperations.** International collaborators on the original project have been Dr. Mike Dodds, Dr. Daniel Kudenko and Prof. Richard Paige of the University of York, England, and Dr. Jan Tobias Mühlberg of KU Leuven, Belgium. Prof. Lüttgen visited York in September 2015 and again in April 2016 to explore various machine learning techniques suitable for use in DSI with Dr. Kudenko, and to discuss approaches for using DSI's output for formal program verification with Dr. Dodds and Prof. Paige. A visit by Dr. White and Mr. Rupprecht to Leuven in August 2015 investigated examples of dynamic data structure usages in low-level software, while Dr. Mühlberg's visit to Bamberg in January 2016 helped us to address challenges in moving DSI from analysing source code to analysing binaries, and to identify future work on our DSI tool for informing the VeriFast verifier. Research on a proof-of-concept for applying DSI to the analysis of malware was initiated at a visit of Prof. Lüttgen and Mr. Rupprecht to Leuven in May 2016.

In addition, we have started a new collaboration with the team of Prof. Herbert Bos, VU Amsterdam, The Netherlands, in Spring 2016. His group specialises in reverse engineering and software analysis for computer security. Their tool Howard is a dynamic excavator for reverse engineering and infers type information from binaries. Two visits by Mr. Rupprecht to Amsterdam in March 2016 and July 2016, resp., as well as a visit by Prof. Bos' then PhD student Mr. Xi Chen to Bamberg in May/June 2016 have successfully prototyped the interfacing of Howard to DSI in the context of porting DSI to binary analysis, for which Howard's type inference capabilities played an important part of the investigation.

The above collaborators have also co-authored publications on the project: Leuven co-authored publications [PSP3, PSP5], York co-authored publication [PSP3], and Amsterdam co-authored publication [PSP5].

**Future cooperations.** In the follow-up project proposed here, the project team will expand the cooperation with KU Leuven and continue the recently established cooperation with VU Amsterdam. The project team will also keep ties with Dr. White who has been the Senior RA on the project until August 2016.

*Distributed Systems and Computer Networks Group, KU Leuven, Belgium:* The Distributed Systems and Computer Networks Group (DistriNet) is an internationally leading research group within the Computer Science department at the Katholieke Universiteit Leuven. It is headed by Prof. Frank Piessens and closely connected to the independent iMinds research institute founded in 2004 by the Flemish government to stimulate innovation in the field of ICT. Research at iMinds-DistriNet focuses on a wide range of problems including topics in distributed and embedded

systems as well as computer security. Dr. Mühlberg is a postdoctoral researcher at DistriNet and will contribute to the follow-up project with his extensive experience in verifying system-level software with VeriFast and in reverse engineering and analysing malware (WPs II/1-4 & III/1-3). He will be joined by Leuven's PhD student Mr. Rafique who specialises in malware analysis and classification (WP II/3).

*Systems and Network Security Group, VU Amsterdam, The Netherlands:* The Systems and Network Security Group (VUSec) at Vrije Universiteit Amsterdam is an internationally leading research group within the Computer Science department at VU and headed by Prof. Herbert Bos. The group's research covers all aspects of system-level security and reliability, including binary and malware analysis and reverse engineering. Of particular interest to us is Prof. Bos' expertise in reverse engineering and extracting type information from binaries (WPs I/2 & II/1-3). The follow-up project will also benefit from Dr. Rawat who is a postdoctoral researcher at VUSec, and his knowledge in dynamic security program analysis and malware classification (WPs II/2-3).

*Further cooperation partner:* Dr. White has been the main innovator of the DSI approach and is the architect of the DSI tool. For personal reasons he is now living in Taiwan, but he is planning to finish his Habilitation at Bamberg on a topic closely related to the DSI project. He will be available as a consultant to the follow-up project and will contribute, in particular, to the work packages related to extending/deepening the DSI approach (WPs I/1-6).

### 5.4.2 Past Collaborators

Within the past three years, the applicants have collaborated scientifically with the following international and national (outside the University of Bamberg) researchers: Prof. Herbert Bos, VU Amsterdam, The Netherlands; Mr. Ferenc Bujtor, U. Augsburg, Germany; Dr. Benoît Caillaud, INRIA Rennes/IRISA, France; Mr. Xi Chen, VU Amsterdam, The Netherlands; Prof. Gianfranco Ciardo, U. Iowa, USA; Prof. Rance Cleaveland, U. Maryland, USA; Dr. Mike Dodds, U. York, England; Prof. Keijo Heljanko, Aalto Univ., Finland; Dr. Daniel Kudenko, U. York, England; Dr. Jan Tobias Mühlberg, KU Leuven, Belgium; Prof. Richard Paige, U. York, England; Prof. Frank Piessens, KU Leuven, Belgium; Prof. Walter Vogler, U. Augsburg, Germany.

## 5.5 Scientific Equipment

All equipment necessary for carrying out the proposed project, i.e., PCs/laptops with standard software, headsets and webcams, will be provided by the University of Bamberg. Funds are requested in Sec. 4.1.2.1 to purchase licenses of the commercial software IDA Pro, which is needed to carry out work packages WP II/1, II/2 and II/4a.

## 5.6 – 5.7 Not Applicable

5.6 Project-relevant cooperation with commercial enterprises; 5.7 Project-relevant participation in commercial enterprises.

# 6 Additional Information: *Not applicable*