# Generating Inductive Shape Predicates for Runtime Checking and Formal Verification*

Jan H. Boockmann[1], Gerald Lüttgen[1], and Jan Tobias Mühlberg[2]

[1] Software Technologies Research Group, University of Bamberg, D-96045 Germany
{jan.boockmann, gerald.luettgen}@swt-bamberg.de
[2] imec-DistriNet, KU Leuven, Celestijnenlaan 200A, B-3001 Belgium
jantobias.muehlberg@cs.kuleuven.be

**Abstract.** Knowing the shapes of dynamic data structures is key when formally reasoning about pointer programs. While modern shape analysis tools employ symbolic execution and machine learning to infer shapes, they often assume well-structured C code or programs written in an idealised language. In contrast, our *Data Structure Investigator* (DSI) tool for program comprehension analyses concrete executions and handles even C programs with complex coding styles.
Our current research on memory safety develops ways for DSI to synthesise inductive shape predicates in separation logic. In the context of trusted computing, we investigate how the inferred predicates can be employed to generate runtime checks for securely communicating dynamic data structures across trust boundaries. We also explore to what extent these predicates, together with additional information extracted by DSI, can be used within general program verifiers such as *VeriFast*.
This paper accompanies a talk at the ISoLA 2018 track *"A Broader View on Verification: From Static to Runtime and Back"*. It introduces DSI, highlights the above use cases, and sketches our approach for synthesising inductive shape predicates.

## 1 Motivation

Formally reasoning about the memory safety and security of C code executing on processors is a serious challenge, especially when dynamic data structures are involved. The advent of separation logic [18] for modularly specifying heap operations and data structure shapes has sparked a wealth of research in the computer-aided verification of pointer programs. This has led to the development of powerful static verifiers, including shape analysis tools such as *Forester* [9] and *Infer* [6, 7] and program verifiers such as *VeriFast* [17]. While the latter requires significant manual effort in annotating programs with contracts, recent automated shape analysis techniques employ machine learning [4, 25] but assume well-structured code and a well-behaved execution environment. However, these

* Research supported by the German Research Foundation (DFG) under grant *"DSI2: Learning Data Structure Behaviour from Executions of Pointer Programs"* (LU 1748/4-2) and by the Research Fund KU Leuven.

assumptions are violated by software for modern processors that include field-programmable hardware such as Intel's *Broadwell Xeon* or Xilinx' *Zync*, and on processors that provide secure enclaves such as Intel's *SGX* [13].

This motivates us to consider recent advancements in dynamic analysis tools for identifying pointer-based data structures such as linked lists and their interconnections such as parent-child nesting, from concrete execution traces of instrumented source code [1, 12, 24] or even binaries [5, 8, 10, 19]. Among these, the *Data Structure Investigator* (DSI) tool [24] distinguishes itself by a novel heap representation based on so-called *strands*, which can handle even C programs employing complex and sometimes 'dirty' coding styles and pointer operations; see, e.g., the cyclic doubly-linked list (DLL) of the Linux kernel [11]. Indeed, cyclicity is a runtime property that is nontrivial to check with static analysis; the same is true when figuring out whether the two pointers in a DLL struct do indeed contribute to a DLL rather than a binary tree.

While DSI has so far aimed at program comprehension, its inferred data structure information may be useful for generating inductive shape predicates for a variety of use cases that range from runtime checking to formal verification, including the following two:

*Specifying secure wrappers.* Isolation and trust boundaries between software and hardware components play an important role when implementing security features, e.g., using trusted execution environments such as *SGX* [13] or *Sancus* [16]. Trust boundaries must be crossed to transfer data in and out of protection domains, thus rendering these transfers a potential attack vector. When linked data structures with pointers are accessed from within an enclave, these pointers may be abused to manipulate the execution flow of the enclave.

The underlying problem can be generalised as execution of trusted code in an untrusted context. It bears similarity with executing formally verified software in an unverified operating system, thereby exposing the verified code to interactions not captured during verification. In a similar way, field-programmable hardware extensions in modern computing systems may violate the assumed program semantics by modifying memory concurrent to the main processor and in a way unknown at the time of software development. It is likely that these problems can best be addressed at runtime.

Recent work [2, 21] has demonstrated how shape specifications of dynamic data structures written in VeriFast's flavour of separation logic can be used to employ *secure wrappers* for copying data between trusted and untrusted system components. These wrappers execute checks of shape properties at runtime, thereby monitoring the secure communication of dynamic data structures between protection domains and helping one to prevent crashes and a range of vulnerabilities including code injection attacks. However, shape specifications are difficult to obtain in practice, not at least due to the serious cost in terms of person hours for developing them when complex, low-level C code is involved [17]. An open question is how to automatically generate shape specifications suitable for secure wrapper synthesis from source code.

*Generating verification annotations.* Our second use case concerns the challenge of formally verifying the memory safety of C programs in tools such as VeriFast [17]. While VeriFast has been successfully employed to verifying C source code of industrial projects, it requires a skilled engineer to annotate each C function with its contract, i.e., a pre- and post-condition specified in a separation logic dialect. Our prior work [15] employed DSI's predecessor *dsOli* [23] to support this time-consuming task, by inferring those parts of contracts that involve data structure shape only.

The tool dsOli combines machine learning and pattern matching to automatically locate and identify operations on linked-list data structures in C programs, and outputs a set of instantiated operation templates. Each such template describes a data structure operation performed by the program, e.g., list inserts and removals. Corresponding verification annotation templates for VeriFast can then be instantiated and injected into the program's source code automatically, which allows VeriFast to discharge memory safety properties either automatically or after slight manual adjustments of the annotations.

However, our approach inherited dsOli's restrictions of well-structured C code and non-nested list structures. The advancements of DSI in terms of its fine-granular strand abstraction of list structures and its robustness against different C coding styles now allows us to generate verification annotations for general list structures, i.e., without the limitation that each shape requires us to define a new template. The research questions here are (a) in how far one can also auto-generate suitable loop invariants and lemmas needed by VeriFast, and (b) to what degree memory safety proofs can be automated in VeriFast. Notably, VeriFast predicates, lemmas and invariants must not only be inferred but phrased in ways that enable VeriFast's advanced automation capabilities.

*Agenda.* Our current research explores to what extent the data structure information excavated by DSI can be employed to address the above uses cases. At the heart will be a novel generator that automatically synthesises inductive shape predicates of linked-list data structures, taking DSI's global strand representation as input and providing local separation logic predicates suitable for VeriFast as output. In this context, we also need to extract some additional information internally inferred by DSI, in order to obtain shape contracts for the functions contained in the C source code under study, in terms of pre- and post-conditions and accompanying lemmas.

Such shape contracts are exactly the input needed to construct secure wrapper functions for protection domains in our first use case. Our second use case of memory safety verification in VeriFast requires the synthesis of more general lemmas and verification annotations. This is significantly more challenging, e.g., due to the necessity of loop invariants, so that we expect to only be able to generate skeletons of verification annotations.

*Organisation.* The remainder of this paper is structured as follows. The next section briefly introduces our DSI tool and its strand abstraction of list-based dynamic data structures. We then explain how shape specifications in VeriFast's

language may be inferred from DSI's output, illustrate this via a simple but non-trivial example involving a Linux-style list with nested child lists, and discuss the challenges for making our approach work for our two use cases.

## 2 DSI: Data Structure Investigator

DSI is a dynamic analysis for the automatic identification of pointer-based data structures [24]. It detects (cyclic) singly and doubly linked lists (SLLs/DLLs) and binary trees, as well as other structures such as skip lists that are not handled by related work [5, 8, 10]. Additionally, DSI allows for arbitrary parent-child nesting combinations of such data structures, which is also out of scope of related work. This section surveys DSI's technologies and sketches its representation of data structure shapes by means of an example.

*Memory abstraction.* DSI operates by executing C source code that is instrumented for recording pointer writes and memory allocations and deallocations. Its dynamic analysis relies on a novel memory abstraction, *strand graphs*, for interpreting the points-to graphs constructed from such recordings. Intuitively, a strand graph represents a data structure shape at a specific time step of the program's execution. Its nodes are strands that essentially consist of singly linked lists, and its edges are strand connections that represent, e.g., nesting.

A *strand* consists of list nodes that are permitted to cover sub-regions of memory and are termed *cells*. A strand's cells must all have the same linkage condition, i.e., all pointers originate at the same linkage offset relative to the cell's start address, and must point to the start address of the following cell. This general definition allows us to deal with complex coding styles of lists in C, such as the one employed by the Linux cyclic DLL [11] whose nodes may run through types of structs that are embedded in outer structs. Thereby, different list nodes may be of different types, and a node's successor (predecessor) field may point inside the successor's (predecessor's) node.

A *strand connection* describes exactly one way out of the potentially multiple ways in which the cells of two strands may be related. Typical connections are, e.g., *indirect nesting* or *overlay nesting*, where the latter means that a child list's head node is contained in its parent node, or the *dll* connection, where two strands running through list nodes are overlaid in such a way that two neighbouring nodes are mutually linked.

*Evidence gathering.* The biggest challenge for detecting linked-list data structures at runtime comes with data structure operations. These tend to temporarily break a data structure's shape, e.g., when rewriting pointers during the insertion into a linked list. To discriminate against degenerate shapes, DSI uses a unique evidence-collecting algorithm, which utilises the structural complexity of an observed shape as evidence measure and which reinforces evidence counts by exploiting structural and temporal repetition.

For structural repetition, DSI detects and folds all strands that perform the same role within one time step, e.g., the strands representing the child lists of

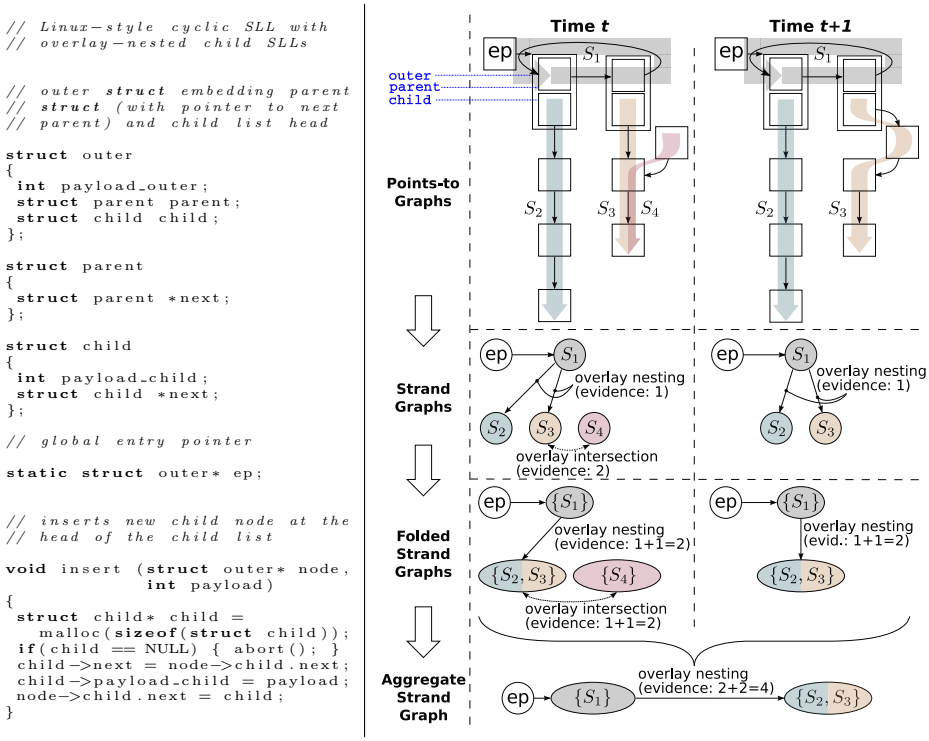


**Fig. 1.** Example: C source code (left) and DSI's strand representation (right).

the nodes of a parent list. Regarding temporal repetition, DSI determines which folded strands represent the same data structure building block over multiple time steps. For each entry pointer to a data structure and each time step in which the entry pointer exists, the temporal repetition is performed by extracting the folded strand graph's subgraph that is reachable from the entry pointer. All extracted subgraphs are then merged over the lifetime of the entry pointer, resulting in an aggregate strand graph.

All evidence counts of strand connections are accumulated when folding strands and aggregating strand graphs. The identified data structure shape then manifests itself by the aggregate strand graph when ignoring strand connections with low evidence counts. Evidence counts for the correct shape of a data structure accumulate quickly, because the majority of stable shapes within a program execution overrides the minority of degenerate shapes. Experimental evaluations with our DSI tool have shown that a significant variety and quantity of complex C source code can be handled and that, in each case, our evidence-based approach leads to the correct identification of data structure shape [24] and reveals sufficient contextual data to inform static verification [15].

*Example.* Fig. 1 contains a simple example of a 'Linux-style' cyclic SLL with overlay-nested non-cyclic child SLLs. A snippet of the C source code is depicted on the left, showing the data structure definition via nested structs and a node insert operation into the child list. DSI's interpretation of the data structure – from a given entry pointer, here `ep` – is displayed on the right in terms of the points-to graphs, the strand graphs and the folded strand graphs across two consecutive time steps $t$ and $t+1$, as well as the resulting aggregate strand graph, each decorated with evidence counts for the detected strand connections.

The time steps are selected such that the data structure is in a degenerate shape at time step $t$, as it is in the middle of an insert operation at the head of the second child list, whereas it is in a stable shape at time step $t+1$, i.e., at the end of the insertion operation. DSI's evidence counts for the strand connection types occurring in this example are as follows: 1 for two nested strands where the head cell of the child strand is located in the same node as the parent strand's cell (*overlay nesting*), and $k$ for two strands that intersect in $k$ nodes (*overlay intersection*). Evidences of corresponding strand connections are simply summed up when folding and again when aggregating strand graphs. Observe that, when aggregating the folded strand graphs, strand $\{S_4\}$ is not considered as the full strand is not reachable from the entry pointer `ep`.

## 3 Inferring Shape Information for VeriFast

To enable the two use cases for DSI envisaged in Sec. 1, a tool that infers shape information for VeriFast on the basis of DSI's analysis results is required. This section first sketches the synthesis of inductive shape predicates by means of our example. We then discuss the concrete challenges for generating secure wrapper specifications and, resp., more general verification annotations.

```
1  struct parent {
2    struct parent *next;
3    //@ struct outer *ghost_outer;
4  };
5
6  predicate NodesCSLL(struct parent *node, struct outer *outer,
7    struct parent *head, int count;) =
8      node == NULL ? count == 0 && head == NULL
9      : count >= 1 && head != NULL
10       &*& [1/2]node->ghost_outer |-> outer
11       &*& node->next |-> ?tail
12       &*& outer->payload_outer |-> _                    // payload of outer
13       &*& outer->child.next |-> ?child                  // entry pointer to child
14       &*& SLL(child, outer, _)                          // child list predicate
15       &*& outer->child.payload_child |-> _              // payload of child
16       &*& malloc_block_outer(outer)                     // allocated memory chunk
17       &*& count == 1 ? tail == head :                   // tail points to head
18           [1/2]tail->ghost_outer |-> ?tail_outer  &*&
19           NodesCSLL(tail, tail_outer, head, count-1);  // continued tail
20
21 predicate CSLL(struct parent *list, struct outer *outer, int count;) =
22   [1/2]list->ghost_outer |-> outer
23   &*& NodesCSLL(list, outer, list, count);
```

**Fig. 2.** Example (cont'd): Inferred VeriFast predicates (*simplified*) for the parent CSSL. Elements coloured in blue, green and red relate to the cyclicity property, the embedded child SLL and the Linux-style list structure of this CSLL, resp.

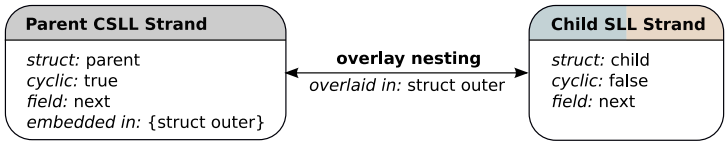


**Fig. 3.** Example (cont'd): DSI's aggregate strand graph with detailed attributes of strands and strand connections.

*Constructing shape predicates from aggregate strand graphs.* Shape predicates are constructed via a two step approach: first, the rich type information available in the program's source code is statically analysed to build a predicate skeleton, and second, the runtime information provided by DSI in form of an aggregate strand graph is employed to refine these skeletons.

The first step creates a predicate skeleton for each struct 'touched' by a strand, using an approach similar to [14] and exploiting the structural similarity between C struct definitions and VeriFast predicate definitions. The skeleton generated for `struct parent` of our example of Fig. 1 consists of the black coloured parts of `predicate NodesCSLL` shown in Fig. 2: parameter `node` is the entry pointer to the list, while `count` is the number of nodes contained in the list. The predicate is inductive, with the empty list being handled as base case in l. 8 and the induction step's recursive call wrt. the tail list in l. 19.

Observe that the skeleton still lacks information regarding the parent list's cyclicity, the nesting of the child list, and the fact that the parent is a Linux-style list. This information cannot be determined statically from the `struct` definitions, but is inferred by DSI's analysis. Indeed, when looking at the details of the aggregate strand graph output by DSI as shown in Fig. 3, one can see that the parent strand has property *cyclic*, contains an `overlaid`-nested child strand of `struct child` nodes, and the parent nodes of type `struct parent` are embedded in `struct outer`. These pieces of information are successively implanted into `predicate NodesCSLL` as follows.

Firstly, the predicate is refined regarding the list's cyclicity property by adding the parts coloured in blue. In particular, `NodesCSLL` gains a third parameter (l. 7), namely the `head` pointer referring to the node in which the cycle is expected to be closed (l. 17). The wrapper `predicate CSLL` invariantly sets the head to the list's original entry pointer `list` (l. 21-23). Secondly, the green coloured parts are added to `predicate NodesCSLL` so as to reflect nesting. Predicate `SLL` (l. 14) describing the child list is simpler than `CSLL` and thus not shown in Fig. 2. However, note that parameter `outer` is required due to the nesting not being indirect nesting but overlay nesting. Thirdly, the parent's Linux-style list is modelled in VeriFast by the red coloured parts in `predicate NodesCSLL`. We first add a new pointer `outer` as parameter to `NodesCSLL` (l. 6) and extend `struct parent` with a *ghost field* `ghost_outer` to express the link between `struct parent` and `struct outer` (l. 3, and similarly in `struct child`). This field is set via *ghost statements* placed into the source code at locations that can

be determined automatically. The connection between a `parent` struct referenced by `node->next` and the corresponding `outer` struct (i.e., `&(outer.parent) == node->next`) is then maintained by consuming a fractional permission `[1/2]` of the heap chunk associated with this reference in `NodesCSLL(node, ...)`, while the other `[1/2]` of that chunk is consumed in `NodesCSLL(tail, ...)` where `tail` is `node->next`. This way, each element of the recursive list definition maintains a partial and unmodifiable reference to its successor, while 'opening' the predicates for both the current `node` and the successor `tail` yields a complete reference that allows for list manipulations at that location. To the best of our knowledge, a Linux-style list has not been verified in VeriFast so far.

To conclude, the predicates obtained for our example constructively represent a fairly intricate data structure and are sufficient to generate or verify code that iterates over the list or modifies it by inserting and removing elements at arbitrary positions. Also note that our approach is not compositional: simply defining an inductive VeriFast predicate for each strand of DSI's inferred strand graph and then gluing the predicates together according to the strand connections does, in general, not result in a shape predicate suitable for VeriFast.

*Generating secure wrapper specifications.* Our first use case employs our approach to synthesising inductive shape predicates for generating secure wrapper specifications for a variant of VeriFast [2, 21]. Essentially, secure wrapper specifications are shape contracts between trusted and untrusted program functions, so that the actual shapes of dynamic data structures passed across trust boundaries can be checked against expected shapes by inspecting or deep-copying the data structures at runtime. This poses an exciting application for DSI at the intersection of formal specification and run-time monitoring, in particular because no existing code needs to be verified.

For the integration with VeriFast, only *precise* predicates may be used, where the same input arguments represent the same memory region and always have the same output arguments. This enables constructive reasoning in VeriFast and, thus, allows for advanced automatic processing in verification and code generation. Because our specifications and wrappers can operate on data structures without taking the functional properties of a program's context into account, e.g., we can traverse or copy a linked list from start to end rather than support insert and remove operations at arbitrary positions, we consider this synthesis to be significantly easier than actual program verification. Relying on extended features of DSI, specifically the identification of entry pointers and list types in function prototypes, we can construct shape contracts that correctly associate C-function parameters with data structures, which we believe can be transformed into wrappers for real application code.

*Generating verification annotations.* Our second use case aims at extending our previous work on inferring shape annotations [15] to support full-program verification in VeriFast. Such annotations should be derived directly from strand graphs by exploiting type and allocation information as well as strand connections, rather than relying on the instantiation of annotation templates. For

each function contract in the program under analysis, DSI will collect the pair of aggregate strand graphs wrt. the function's head and return, together with the entry pointers for any of its pointer parameters and locally declared function pointers that are relevant to the program's dynamic data structure. Overall, this will result in a fairly generic approach to annotation generation, which would be capable of providing partial annotations for programs that make use of data structures for which no annotation templates exist.

However, generating verification annotations is a much more difficult challenge than generating shape contracts for secure wrappers. Firstly, program verification requires stronger shape contracts that, e.g., express whether and how a data structure is modified, for which list length is an important information. Indeed, DSI internally stores the length of strands so that we can infer whether a strand has grown or shrunk between some function call and return, or between consecutive iterations of a loop body. This provides evidence as to whether the operation encapsulating a function is, e.g., an insert or a delete operation, and which strand is traversed in a particular loop via which entry pointer.

Secondly, we also have to provide annotations that facilitate the verification of function bodies. In particular, we aim to construct loop invariants, which typically requires additional specification elements such as supporting lemmas or even inline annotations to handle data structure manipulations that cannot be expressed as precise predicates. Early-out conditions in iterators as well as insert and delete operations at arbitrary positions within a data structure are challenges that are considered to be hard for verification tools. Here, we believe that our annotation inference approach together with extended automation in VeriFast [14, 22] can alleviate verification engineers from some of the burden of writing program specifications. However, it remains to be seen how much can be done in this respect, particularly when considering that the inferred annotations must be phrased in a way that enables VeriFast's automation capabilities.

## 4 Outlook

Knowing the shapes of a program's dynamic data structures is essential when reasoning about pointer programs. In the context of low-level C programs that frequently employ complex coding styles, this paper argued that such shapes can be inferred automatically by the recent dynamic shape analyser DSI [24] and represented in the separation logic dialect of the VeriFast verifier [17]. This potentially enables two important use cases in the context of secure and safe computing: the automatic synthesis of secure wrappers for securing trust boundaries at runtime [21] and the formal verification of memory safety properties [17].

While our current work focuses on single-threaded C source code, it is conceivable that our applications of DSI to runtime checking and formal verification may be extended to (i) multi-threaded code by considering the *VerCors* verifier [3] and (ii) C/C++ binaries by employing the *DSIbin* front end of DSI [19]. Both extensions are worthwhile because security applications frequently involve concurrent computing architectures and untrusted components

that are only available in compiled form. However, the adaptation of DSI to binaries requires either a novel instrumentation tool for modern processors with field-programmable hardware extensions, or extending DSI with memory snapshot support [20] to correlate strand graphs between non-contiguous time steps.

## References


1. 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 *Software Visualization (SOFTVIS '10)*, pages 53–62. ACM, 2010.
2. P. Agten, B. Jacobs, and F. Piessens. Sound modular verification of C code executing in an unverified context. In *Principles of Programming Languages (POPL '15)*, pages 581–594. ACM, 2015.
3. S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The VerCors tool set: Verification of parallel and concurrent software. In *Integrated Formal Methods (iFM '17)*, volume 10510 of *LNCS*, pages 102–110. Springer, 2017.
4. M. Brockschmidt, Y. Chen, P. Kohli, S. Krishna, and D. Tarlow. Learning shape analysis. In *Static Analysis Symposium (SAS '17)*, volume 10422 of *LNCS*, pages 66–87. Springer, 2017.
5. J. Caballero, G. Grieco, M. Marron, Z. Lin, and D. Urbina. Artiste: Automatic generation of hybrid data structure signatures from binary code executions. Technical Report TR-IMDEA-SW-2012-001, IMDEA, Spain, 2012.
6. C. Calcagno and D. Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NASA Formal Methods Symposium (NFM '11)*, volume 6617 of *LNCS*, pages 459–465. Springer, 2011.
7. C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NASA Formal Methods Symposium (NFM '15)*, volume 9058 of *LNCS*, pages 3–11. Springer, 2015.
8. 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.
9. L. Holík, O. Lengál, A. Rogalewicz, J. Šimáček, and T. Vojnar. Fully automated shape analysis based on forest automata. In *Computer Aided Verification (CAV '13)*, volume 8044 of *LNCS*, pages 740–755. Springer, 2013.
10. C. Jung and N. Clark. DDT: Design and evaluation of a dynamic program analysis for optimizing data structure usage. In *Microarchitecture Symposium (MICRO '09)*, pages 56–66. ACM, 2009.
11. Linux kernel 4.1 Cyclic DLL (`include/linux/list.h`). http://www.kernel.org/. Accessed: 31st January 2017.
12. 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.
13. F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Hardware and Architectural Support for Security and Privacy (HASP '13)*, page 10. ACM, 2013.
14. M. Mohsen and B. Jacobs. One step towards automatic inference of formal specifications using automated VeriFast. In *Critical Systems: Formal Methods and*

*Automated Verification (FMICS-AVoCS '16)*, volume 9933 of *LNCS*, pages 56–64. Springer, 2016.

15. J. T. Mühlberg, D. H. White, M. Dodds, G. Lüttgen, and F. Piessens. Learning assertions to verify linked-list programs. In *Software Engineering and Formal Methods (SEFM '15)*, volume 9276 of *LNCS*, pages 37–52. Springer, 2015.
16. J. Noorman, J. van Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. Freiling. Sancus 2.0: A low-cost security architecture for IoT devices. *ACM Transactions on Privacy and Security*, 20(3):7:1–7:33, 2017.
17. P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens. Software verification with VeriFast: Industrial case studies. *Science of Computer Programming*, 82:77–97, 2014.
18. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS '02)*, pages 55–74. IEEE, 2002.
19. T. Rupprecht, X. Chen, D. H. White, J. H. Boockmann, G. Lüttgen, and H. Bos. DSIbin: Identifying dynamic data structures in C/C++ binaries. In *Automated Software Engineering (ASE '17)*, pages 331–341. IEEE/ACM, 2017.
20. D. Urbina, Y. Gu, J. Caballero, and Z. Lin. Sigpath: A memory graph based approach for program data introspection and modification. In *Research in Computer Security (ESORICS '14)*, volume 8713 of *LNCS*, pages 237–256. Springer, 2014.
21. N. van Ginkel, R. Strackx, and F. Piessens. Automatically generating secure wrappers for SGX enclaves from separation logic specifications. In *Programming Languages and Systems (APLAS '17)*, volume 10695 of *LNCS*, pages 105–123. Springer, 2017.
22. F. Vogels, B. Jacobs, F. Piessens, and J. Smans. Annotation inference for separation logic based verifiers. In *Formal Methods for Open Object-based Distributed Systems (FMOODS '11)*, volume 6722 of *LNCS*, pages 319–333. Springer, 2011.
23. D. H. White and G. Lüttgen. Identifying dynamic data structures by learning evolving patterns in memory. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '13)*, volume 7795 of *LNCS*, pages 354–369. Springer, 2013.
24. D. H. White, T. Rupprecht, and G. Lüttgen. DSI: An evidence-based approach to identify dynamic data structures in C programs. In *Software Testing and Analysis (ISSTA '16)*, pages 259–269. ACM, 2016.
25. H. Zhu, G. Petri, and S. Jagannathan. Automatically learning shape specifications. In *Programming Language Design and Implementation (PLDI '16)*, pages 491–507. ACM, 2016.