Verifying Linked-List Programs and Beyond

Jan Tobias Mühlberg

iMinds-DistriNet, KU Leuven, Celestijnenlaan 200A, B-3001 Belgium
jantobias.muehlberg@cs.kuleuven.be

dsOli Day @ Bamberg, January 2016

VeriFast [JSP+11]

Fast, sound and modular verification tool for C and Java

VeriFast [JSP+11]

Fast, sound and modular verification tool for C and Java

Checks that

- Program executes with no runtime exceptions
- Function contracts hold for each execution

VeriFast [JSP+11]

Fast, sound and modular verification tool for C and Java

Checks that

- Program executes with no runtime exceptions
- Function contracts hold for each execution

Contracts: source code annotations in separation logic.

VeriFast [JSP+11]

Fast, sound and modular verification tool for C and Java

Checks that

- Program executes with no runtime exceptions
- Function contracts hold for each execution

Contracts: source code annotations in separation logic.

```
1
2
3
4
5  void push(ElementType X, Stack S)
6
7
8  {
9    Stack TmpCell;
10   TmpCell = malloc(sizeof(struct ↩
         Node));
11
12   TmpCell->Element = X;
13   TmpCell->Next = S->Next;
14   S->Next = TmpCell;
15 }
```

VeriFast [JSP+11]

Fast, sound and modular verification tool for C and Java

Checks that

- Program executes with no runtime exceptions
- Function contracts hold for each execution

Contracts: source code annotations in separation logic.

```
1
2
3
4
5  void push(ElementType X, Stack S)
6  //@ requires SLL(S, ?n_0;
7  //@ ensures SLL(S, ?n_0+1;
8  {
9    Stack TmpCell;
10   TmpCell = malloc(sizeof(struct ↩
         Node));
11
12   TmpCell->Element = X;
13   TmpCell->Next = S->Next;
14   S->Next = TmpCell;
15 }
```

VeriFast [JSP+11]

Fast, sound and modular verification tool for C and Java

Checks that

- Program executes with no runtime exceptions
- Function contracts hold for each execution

Contracts: source code annotations in separation logic.

```
1  /*@ predicate SLL(struct Node *↩
       list, int count) =
2    &*& list->Next |-> ?next
3    &*& SLL(next, count - 1); @*/
4
5  void push(ElementType X, Stack S)
6  //@ requires SLL(S, ?n_0;
7  //@ ensures SLL(S, ?n_0+1;
8  {
9    Stack TmpCell;
10   TmpCell = malloc(sizeof(struct ↩
       Node));
11
12   TmpCell->Element = X;
13   TmpCell->Next = S->Next;
14   S->Next = TmpCell;
15 }
```

From dsOli Templates to Separation Logic [MWD+15]

From dsOli Templates to Separation Logic [MWD+15]

[WL13]: Potential operations are labelled by template matching against a repository of manually defined templates for the pre and post states of an operation.

From dsOli Templates to Separation Logic [MWD+15]

[WL13]: Potential operations are labelled by template matching against a repository of manually defined templates for the pre and post states of an operation.

- Templates describe the linkage between elements of a DS (i.e. SLL with a head element) and operations performed on that DS.
- Template matches tell us which LOC are involved in an operation and which data objects and types are involved.

From dsOli Templates to Separation Logic [MWD+15]

[WL13]: Potential operations are labelled by template matching against a repository of manually defined templates for the pre and post states of an operation.

- Templates describe the linkage between elements of a DS (i.e. SLL with a head element) and operations performed on that DS.
- Template matches tell us which LOC are involved in an operation and which data objects and types are involved.

Our Approach

- Use templates and type information to generate predicates that specify the DS as a whole.
- Use LOC information to generate function contracts and proof machinery that specify program behaviour.

From dsOli Templates to Separation Logic [MWD+15]

[WL13]: Potential operations are labelled by template matching against a repository of manually defined templates for the pre and post states of an operation.

- Templates describe the linkage between elements of a DS (i.e. SLL with a head element) and operations performed on that DS.
- Template matches tell us which LOC are involved in an operation and which data objects and types are involved.

Our Approach

- Use templates and type information to generate predicates that specify the DS as a whole.
- Use LOC information to generate function contracts and proof machinery that specify program behaviour.

Let's look at an example.

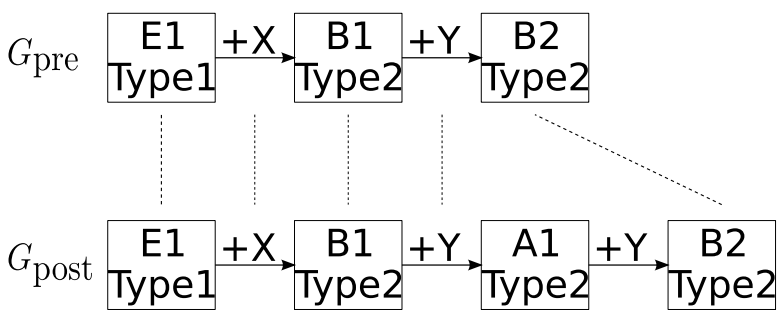
From dsOli Templates to Separation Logic [MWD+15]

```
1  typedef struct Node *Stack;
2
3  struct Node {
4    ElementType  Element;
5    struct Node  *Next;
6  };
7
8
9
10
11
12
13
```

```
14
15
16
17
18
19
20
21 void push(ElementType X, Stack S)
22
23
24 {
25   Stack TmpCell;
26   TmpCell = malloc(
27     sizeof(struct Node));
28   if( TmpCell == NULL ) {
29     printf( "Out of space!" );
30     exit(EXIT_FAILURE);
31   }
32   else {
33
34     TmpCell->Element = X;
35     TmpCell->Next = S->Next;
36     S->Next = TmpCell;
37
38
39   }
40 }
```

From dsOli Templates to Separation Logic [MWD+15]

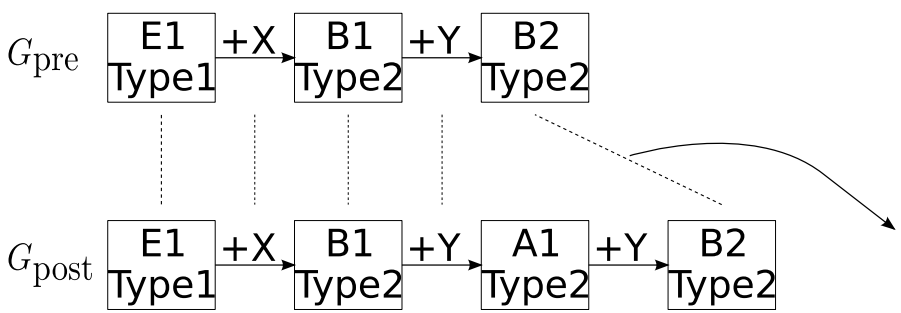
```
1  typedef struct Node *Stack;
2
3  struct Node {
4    ElementType  Element;
5    struct Node  *Next;
6  };
7
8
9
10
11
12
13
```




```
14
15
16
17
18
19
20
21 void push(ElementType X, Stack S)
22
23
24 {
25   Stack TmpCell;
26   TmpCell = malloc(
27     sizeof(struct Node));
28   if( TmpCell == NULL ) {
29     printf( "Out of space!" );
30     exit(EXIT_FAILURE);
31   }
32   else {
33
34     TmpCell->Element = X;
35     TmpCell->Next = S->Next;
36     S->Next = TmpCell;
37
38
39   }
40 }
```

From dsOli Templates to Separation Logic [MWD+15]

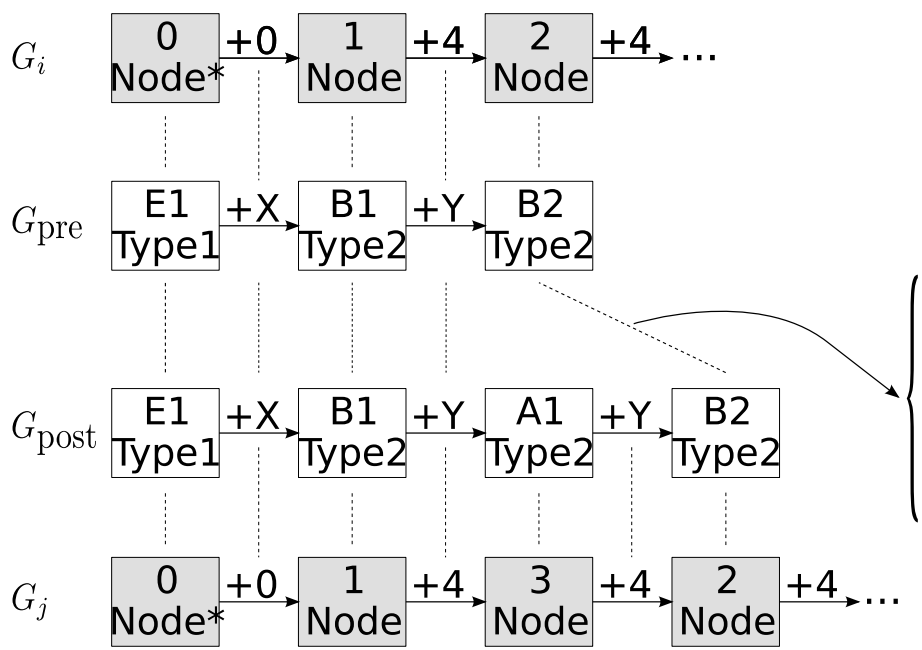
```
1  typedef struct Node *Stack;
2
3  struct Node {
4    ElementType  Element;
5    struct Node  *Next;
6  };
7
8
9
10
11
12
13
```




```
14
15
16
17
18
19
20
21 void push(ElementType X, Stack S)
22
23
24 {
25   Stack TmpCell;
26   TmpCell = malloc(
27     sizeof(struct Node));
28   if( TmpCell == NULL ) {
29     printf( "Out of space!" );
30     exit(EXIT_FAILURE);
31   }
32   else {
33
34     TmpCell->Element = X;
35     TmpCell->Next = S->Next;
36     S->Next = TmpCell;
37
38
39   }
40 }
```

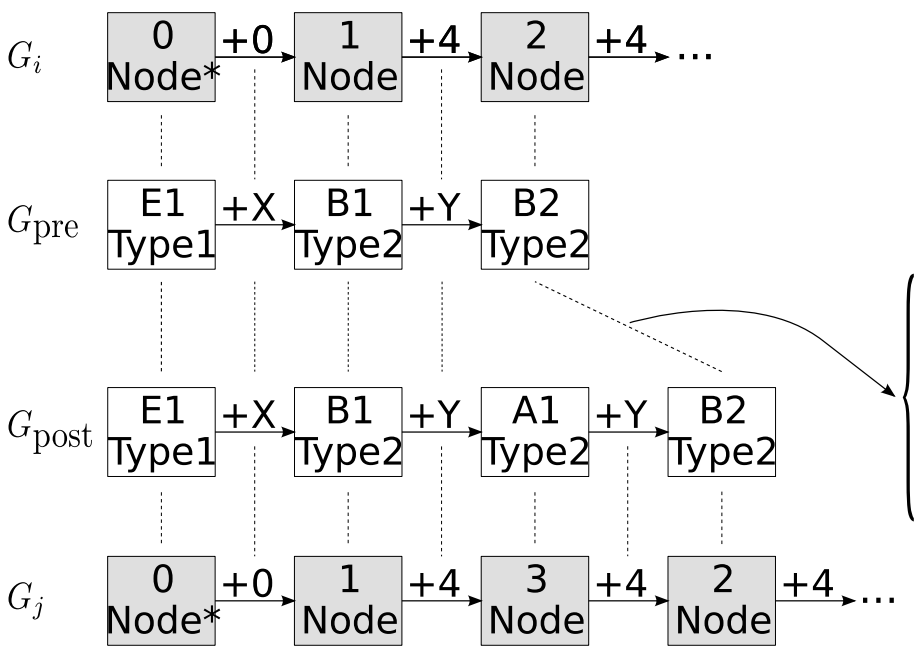
From dsOli Templates to Separation Logic [MWD+15]

```
1  typedef struct Node *Stack;
2
3  struct Node {
4    ElementType  Element;
5    struct Node  *Next;
6  };
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21 void push(ElementType X, Stack S)
22
23
24 {
25   Stack TmpCell;
26   TmpCell = malloc(
27     sizeof(struct Node));
28   if( TmpCell == NULL ) {
29     printf( "Out of space!" );
30     exit(EXIT_FAILURE);
31   }
32   else {
33
34     TmpCell->Element = X;
35     TmpCell->Next = S->Next;
36     S->Next = TmpCell;
37
38
39   }
40 }
```



From dsOli Templates to Separation Logic [MWD+15]

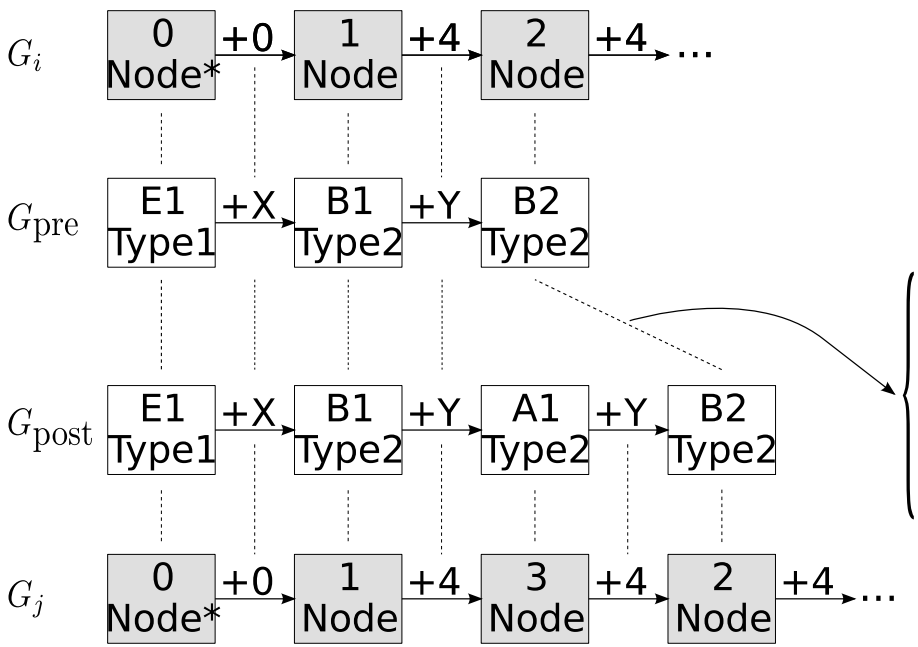
```
1  typedef struct Node *Stack;
2
3  struct Node {
4    ElementType  Element;
5    struct Node  *Next;
6  };
7
8  /*@ predicate SLLNodes_Node(struct Node *↩
       node, int count) =
9    node == 0 ? count==0 : 0<count
10   &*& node->Element |-> _
11   &*& node->Next |-> ?next
12   &*& malloc_block_Node(node)
13   &*& SLLNodes_Node(next,count-1); @*/
```




```
14 /*@ predicate SLL_Node(struct ↩
       Node *list, int count) =
15   0 <= count
16   &*& list->Element |-> _
17   &*& list->Next |-> ?head
18   &*& malloc_block_Node(list)
19   &*& SLLNodes_Node(head, count);↩
       @*/
20
21 void push(ElementType X, Stack S)
22
23
24 {
25   Stack TmpCell;
26   TmpCell = malloc(
27     sizeof(struct Node));
28   if( TmpCell == NULL ) {
29     printf( "Out of space!" );
30     exit(EXIT_FAILURE);
31   }
32   else {
33
34     TmpCell->Element = X;
35     TmpCell->Next = S->Next;
36     S->Next = TmpCell;
37
38
39   }
40 }
```

From dsOli Templates to Separation Logic [MWD+15]

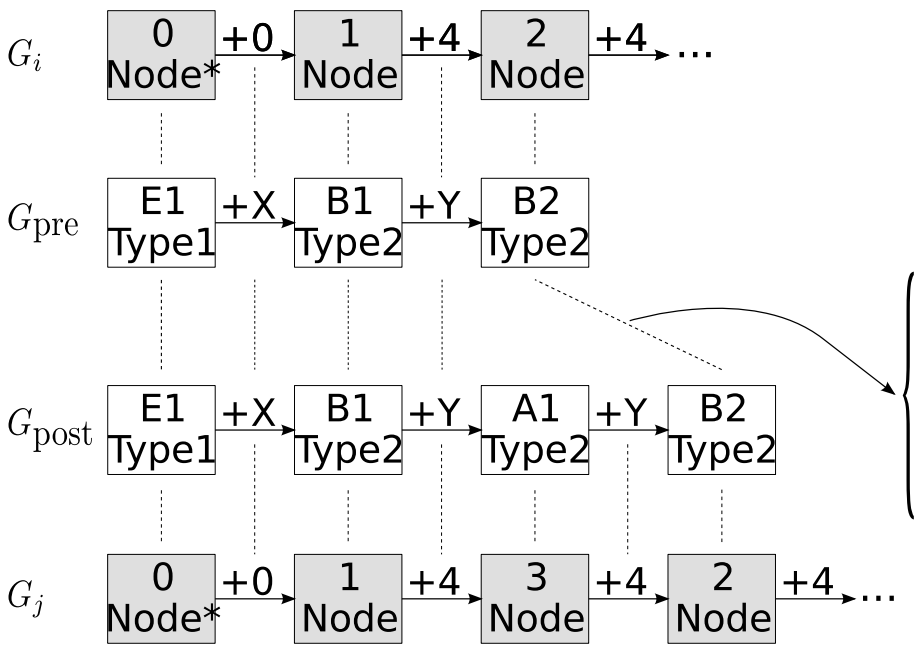
```
1  typedef struct Node *Stack;
2
3  struct Node {
4    ElementType  Element;
5    struct Node  *Next;
6  };
7
8  /*@ predicate SLLNodes_Node(struct Node *↩
       node, int count) =
9    node == 0 ? count==0 : 0<count
10   &*& node->Element |-> _
11   &*& node->Next |-> ?next
12   &*& malloc_block_Node(node)
13   &*& SLLNodes_Node(next,count-1); @*/
```




```
14 /*@ predicate SLL_Node(struct ↩
       Node *list, int count) =
15   0 <= count
16   &*& list->Element |-> _
17   &*& list->Next |-> ?head
18   &*& malloc_block_Node(list)
19   &*& SLLNodes_Node(head, count);↩
       @*/
20
21 void push(ElementType X, Stack S)
22
23
24 {
25   Stack TmpCell;
26   TmpCell = malloc(
27     sizeof(struct Node));
28   if( TmpCell == NULL ) {
29     printf( "Out of space!" );
30     exit(EXIT_FAILURE);
31   }
32   else {
33 //@ open SLL_Node(S, n_0);
34     TmpCell->Element = X;
35     TmpCell->Next = S->Next;
36     S->Next = TmpCell;
37 //@ close SLLNodes_Node(S->Next, ↩
       n_0+1);
38 //@ close SLL_Node(S, n_0+1);
39   }
40 }
```

From dsOli Templates to Separation Logic [MWD+15]

```
1  typedef struct Node *Stack;
2
3  struct Node {
4    ElementType  Element;
5    struct Node  *Next;
6  };
7
8  /*@ predicate SLLNodes_Node(struct Node *↩
       node, int count) =
9    node == 0 ? count==0 : 0<count
10   &*& node->Element |-> _
11   &*& node->Next |-> ?next
12   &*& malloc_block_Node(node)
13   &*& SLLNodes_Node(next,count-1); @*/
```




```
14 /*@ predicate SLL_Node(struct ↩
       Node *list, int count) =
15   0 <= count
16   &*& list->Element |-> _
17   &*& list->Next |-> ?head
18   &*& malloc_block_Node(list)
19   &*& SLLNodes_Node(head, count);↩
       @*/
20
21 void push(ElementType X, Stack S)
22 //@ requires SLL_Node(S, ?n_0);
23 //@ ensures SLL_Node(S, n_0+1);
24 {
25   Stack TmpCell;
26   TmpCell = malloc(
27     sizeof(struct Node));
28   if( TmpCell == NULL ) {
29     printf( "Out of space!" );
30     exit(EXIT_FAILURE);
31   }
32   else {
33 //@ open SLL_Node(S, n_0);
34     TmpCell->Element = X;
35     TmpCell->Next = S->Next;
36     S->Next = TmpCell;
37 //@ close SLLNodes_Node(S->Next, ↩
       n_0+1);
38 //@ close SLL_Node(S, n_0+1);
39   }
40 }
```

From dsOli Templates to Separation Logic [MWD+15]

Is it any good?

We (used to) have a prototypic tool that parses C code and XML analysis output from dsOli, and outputs VeriFast annotations.

- Evaluated on SLLs and DLLs in two of the textbook examples from [WL13], and two list implementations from OSS projects
- Only functions that modify linked data structures are annotated intentionally; stubs are generated for all other functions

From dsOli Templates to Separation Logic [MWD+15]

Is it any good?

We (used to) have a prototypic tool that parses C code and XML analysis output from dsOli, and outputs VeriFast annotations.

- Evaluated on SLLs and DLLs in two of the textbook examples from [WL13], and two list implementations from OSS projects
- Only functions that modify linked data structures are annotated intentionally; stubs are generated for all other functions
- Almost enough for push-button-verification in 3 cases!

From dsOli Templates to Separation Logic [MWD+15]

Is it any good?

We (used to) have a prototypic tool that parses C code and XML analysis output from dsOli, and outputs VeriFast annotations.

- Evaluated on SLLs and DLLs in two of the textbook examples from [WL13], and two list implementations from OSS projects
- Only functions that modify linked data structures are annotated intentionally; stubs are generated for all other functions
- Almost enough for push-button-verification in 3 cases!
- Generated contracts allow for verifying basic safety properties only, but can be extended by the verification engineer.

What Next?

Memory safety is pretty cool, but what we actually want to do goes far beyond it: handling concurrency, race conditions, etc.; functional properties, correct I/O behaviour, cryptographic protocols; security properties, component isolation, information flow.

What Next?

Memory safety is pretty cool, but what we actually want to do goes far beyond it: handling concurrency, race conditions, etc.; functional properties, correct I/O behaviour, cryptographic protocols; security properties, component isolation, information flow.

Case Studies:

- Java Smart Card (eID) [PMP+14]
- Linux drivers and low-level services [PMP+14, PMJSP12, PJP15]
- Polar SSL, Amazon s2n [VJ15]
- Protected Module Architectures ([BNMP15, MNP15])

What Next?

Memory safety is pretty cool, but what we actually want to do goes far beyond it: handling concurrency, race conditions, etc.; functional properties, correct I/O behaviour, cryptographic protocols; security properties, component isolation, information flow.

Case Studies:

- Java Smart Card (eID) [PMP+14]
- Linux drivers and low-level services [PMP+14, PMJSP12, PJP15]
- Polar SSL, Amazon s2n [VJ15]
- Protected Module Architectures ([BNMP15, MNP15])

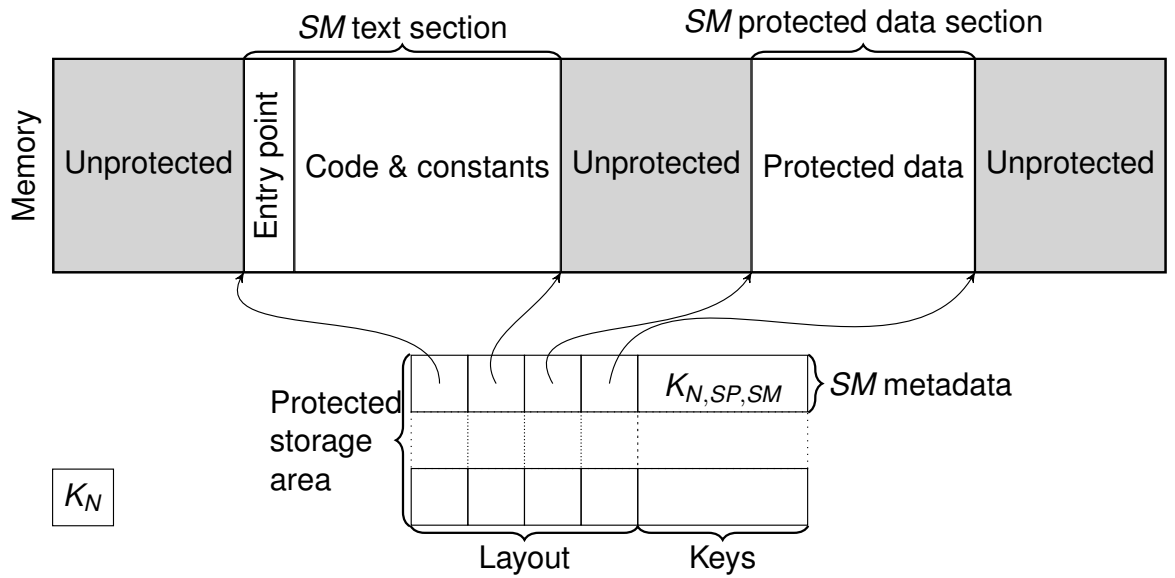
None of these case studies employs data structures beyond SLLs.

Protected Module Architectures

Protected Module Architectures: Sancus [NAD+13]

Sancus enables strong isolation, attestation and secure communication for embedded software components:

- Implements Program Counter Based Access Control [SPP10] for Software Modules (SMs) on single-address-space architectures

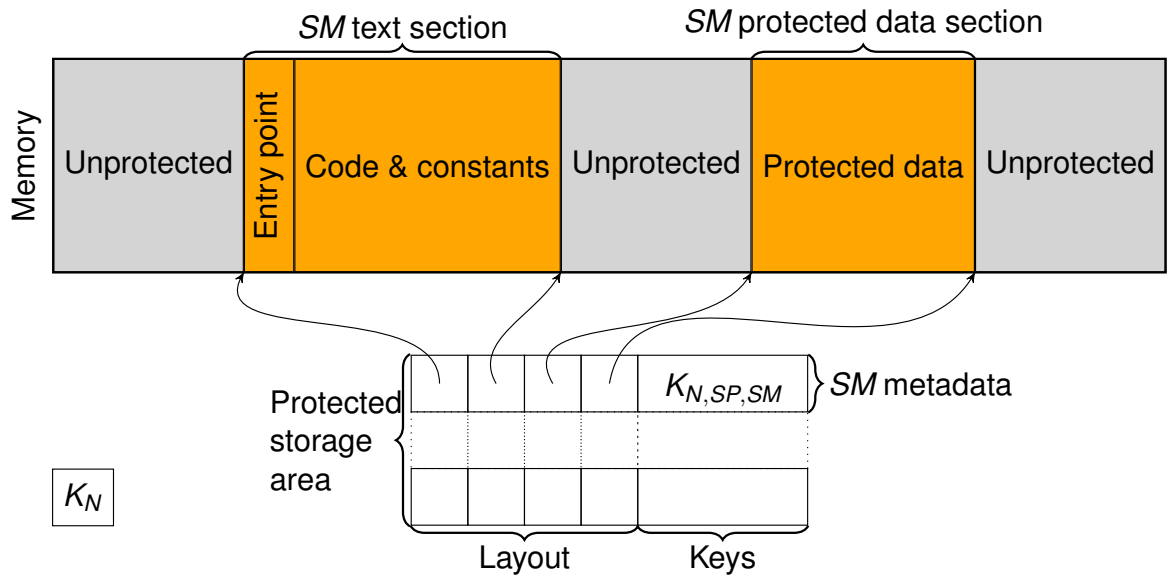


Protected Module Architectures: Sancus [NAD+13]

Sancus enables strong isolation, attestation and secure communication for embedded software components:

- Implements Program Counter Based Access Control [SPP10] for Software Modules (SMs) on single-address-space architectures

Public and protected sections

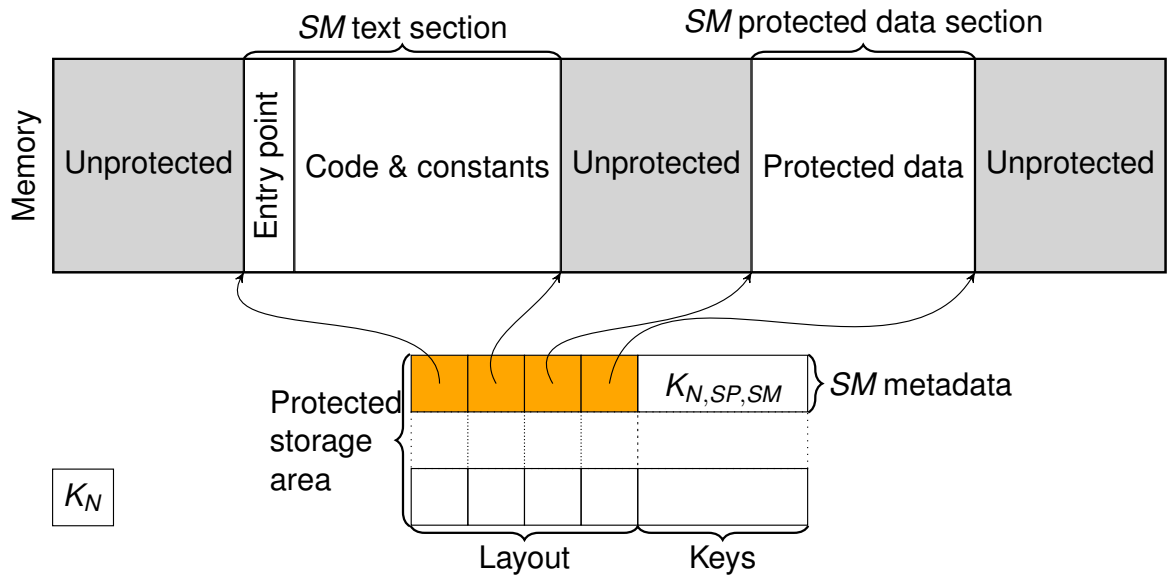


Protected Module Architectures: Sancus [NAD+13]

Sancus enables strong isolation, attestation and secure communication for embedded software components:

- Implements Program Counter Based Access Control [SPP10] for Software Modules (SMs) on single-address-space architectures

Module layout

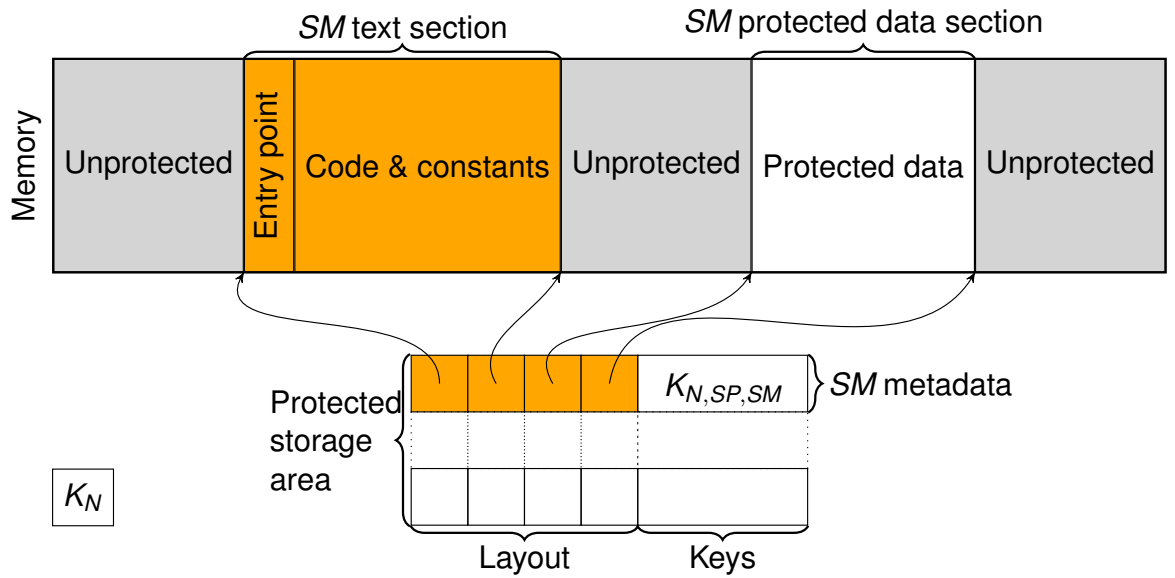


Protected Module Architectures: Sancus [NAD+13]

Sancus enables strong isolation, attestation and secure communication for embedded software components:

- Implements Program Counter Based Access Control [SPP10] for Software Modules (SMs) on single-address-space architectures

Module identity

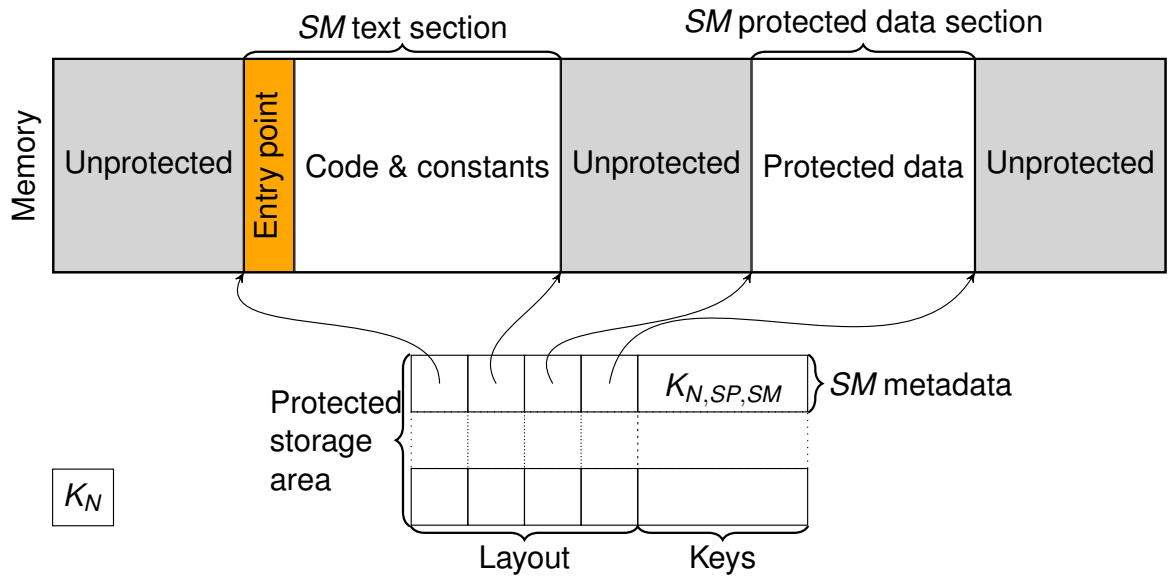


Protected Module Architectures: Sancus [NAD+13]

Sancus enables strong isolation, attestation and secure communication for embedded software components:

- Implements Program Counter Based Access Control [SPP10] for Software Modules (SMs) on single-address-space architectures

Module entry point

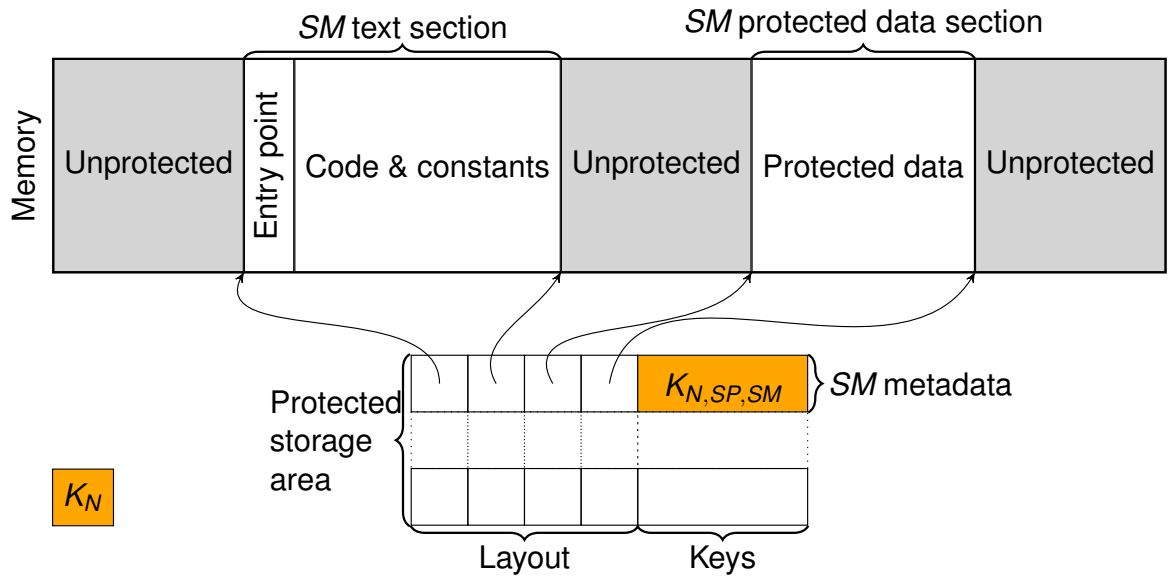


Protected Module Architectures: Sancus [NAD+13]

Sancus enables strong isolation, attestation and secure communication for embedded software components:

- Implements Program Counter Based Access Control [SPP10] for Software Modules (SMs) on single-address-space architectures

Module keys



Protected Module Architectures: Sancus [NAD+13]

Sancus enables strong isolation, attestation and secure communication for embedded software components:

- Implements Program Counter Based Access Control [SPP10] for Software Modules (SMs) on single-address-space architectures
- Provides efficient cryptographic primitives and key handling
- Reference implementation based on the openMSP430

Protected Module Architectures: Sancus [NAD+13]

Sancus enables strong isolation, attestation and secure communication for embedded software components:

- Implements Program Counter Based Access Control [SPP10] for Software Modules (SMs) on single-address-space architectures
- Provides efficient cryptographic primitives and key handling
- Reference implementation based on the openMSP430
- Isolation vs. shared memory communication [BNMP15]

Protected Module Architectures: Sancus [NAD+13]

Sancus enables strong isolation, attestation and secure communication for embedded software components:

- Implements Program Counter Based Access Control [SPP10] for Software Modules (SMs) on single-address-space architectures
- Provides efficient cryptographic primitives and key handling
- Reference implementation based on the openMSP430
- Isolation vs. shared memory communication [BNMP15]

Verification challenges:

Protected Module Architectures: Sancus [NAD+13]

Sancus enables strong isolation, attestation and secure communication for embedded software components:

- Implements Program Counter Based Access Control [SPP10] for Software Modules (SMs) on single-address-space architectures
- Provides efficient cryptographic primitives and key handling
- Reference implementation based on the openMSP430
- Isolation vs. shared memory communication [BNMP15]

Verification challenges:

- Is a given SM memory safe or does it have vulnerabilities?

Protected Module Architectures: Sancus [NAD+13]

Sancus enables strong isolation, attestation and secure communication for embedded software components:

- Implements Program Counter Based Access Control [SPP10] for Software Modules (SMs) on single-address-space architectures
- Provides efficient cryptographic primitives and key handling
- Reference implementation based on the openMSP430
- Isolation vs. shared memory communication [BNMP15]

Verification challenges:

- Is a given SM memory safe or does it have vulnerabilities?
- Does the SM leak secrets?

Protected Module Architectures: Sancus [NAD+13]

Sancus enables strong isolation, attestation and secure communication for embedded software components:

- Implements Program Counter Based Access Control [SPP10] for Software Modules (SMs) on single-address-space architectures
- Provides efficient cryptographic primitives and key handling
- Reference implementation based on the openMSP430
- Isolation vs. shared memory communication [BNMP15]

Verification challenges:

- Is a given SM memory safe or does it have vulnerabilities?
- Does the SM leak secrets?
- Does the SM perform I/O correctly? Does it construct valid cryptographic messages?

Protected Module Architectures: Sancus [NAD+13]

Sancus enables strong isolation, attestation and secure communication for embedded software components:

- Implements Program Counter Based Access Control [SPP10] for Software Modules (SMs) on single-address-space architectures
- Provides efficient cryptographic primitives and key handling
- Reference implementation based on the openMSP430
- Isolation vs. shared memory communication [BNMP15]

Verification challenges:

- Is a given SM memory safe or does it have vulnerabilities?
- Does the SM leak secrets?
- Does the SM perform I/O correctly? Does it construct valid cryptographic messages?
- SM re-entrance, thread safety, real-time behaviour, interaction with other SMs, etc.

Protected Module Architectures: Intuitive Semantics

```
1  DECLARE_SM(count, 0x1234);
2
3  static int SM_DATA(count)
4    counter = 0;
5  static void SM_DATA(count)
6    (*observer)(int);
7
8  void SM_FUNC(count)
9  inc_internal(void) {
10   printf ("%i\n", counter);
11   counter = counter + 1;
12   printf ("%i\n", counter);
13 }
14
15 void SM_ENTRY(count)
16 set_observer(void o(int)) {
17   observer = o;
18 }
19
20 void SM_ENTRY(count)
21 inc() {
22   inc_internal();
23   observer(counter);
24 }
```

```
25 int SM_ENTRY(count)
26 get_count() {
27   return counter;
28 }
29
30 int main(int ac, char** av)
31 {
32   counter = 42;
33   inc_internal();
34   int *c = &counter;
35
36   sancus_enable(&count);
37   counter = 43;
38   set_observer(NULL);
39
40   do {
41     inc();
42   } while (get_count !=
43     SOME_OPCODE - 1)
44   set_observer(c);
45   inc();
46
47   return 0;
48 }
```

Protected Module Architectures: Verification

Interacting Protection Domains

- Similar to threads and locks
- Configurable multi-architecture support
- Labelled heaps

Protected Module Architectures: Verification

Interacting Protection Domains

- Similar to threads and locks
- Configurable multi-architecture support
- Labelled heaps

Information Flow

- Labelled heaps

Protected Module Architectures: Verification

Interacting Protection Domains

- Similar to threads and locks
- Configurable multi-architecture support
- Labelled heaps

Information Flow

- Labelled heaps

Interruption and Re-Entrance

- *Implementation still unclear*

Protected Module Architectures: Verification

Interacting Protection Domains

- Similar to threads and locks
- Configurable multi-architecture support
- Labelled heaps

Information Flow

- Labelled heaps

Interruption and Re-Entrance

- *Implementation still unclear*

Real-Time Behaviour

- Approximation of Worst Case Execution Time
- *Verify a scheduler*

Protected Module Architectures: Verification

Interacting Protection Domains

- Similar to threads and locks
- Configurable multi-architecture support
- Labelled heaps

Information Flow

- Labelled heaps

Interruption and Re-Entrance

- *Implementation still unclear*

Real-Time Behaviour

- Approximation of Worst Case Execution Time
- *Verify a scheduler*

Automation

- Automatic `open`/`close` for precise predicates

Summary & Conclusions

Brief intro to VeriFast [PMP+14].

Presented an extensible approach to automated verification of programs that employ list-based dynamic data structures [MWD+15] in VeriFast, based on dsOli [WL13] output.

There is lots of ongoing research that involves VeriFast, data structures are currently not in the focus.

Security architectures for the IoT can benefit from formal verification (or safe languages). Challenges are with respect to changes in the semantics of C, information flow properties and concurrency.

Current work on the VeriFast tool involves support for different CPU architectures, isolated (labelled) heaps, timing properties, cryptography, and (marginally) on automation.

Thank you!

Thanks for listening!

References I

J. V. Bulck, J. Noorman, J. T. Mühlberg, and F. Piessens.

Secure resource sharing for embedded protected module architectures.
In *WISTP '15*, vol. 9311 of *LNCS*, pp. 71–87, Heidelberg, 2015. Springer.

B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens.

VeriFast: A powerful, sound, predictable, fast verifier for C and Java.
In *NASA Formal Methods 2011*, vol. 6617 of *LNCS*, pp. 41–55, Heidelberg, 2011. Springer.

J. T. Mühlberg, J. Noorman, and F. Piessens.

Lightweight and flexible trust assessment modules for the Internet of Things.
In *ESORICS '15*, vol. 9326 of *LNCS*, pp. 503–520, Heidelberg, 2015. Springer.

J. T. Mühlberg, D. White, M. Dodds, G. Lüttgen, and F. Piessens.

Learning assertions to verify linked-list programs.
In *SEFM '15*, vol. 9276 of *LNCS*, pp. 37–52, Heidelberg, 2015. Springer.

J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens.

Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base.
In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pp. 479–494, Berkeley, CA, USA, 2013. USENIX Association.

W. Penninckx, B. Jacobs, and F. Piessens.

Sound, modular and compositional verification of the input/output behavior of programs.
In *ESOP 2015*, vol. 9032 of *LNCS*, pp. 158–182. Springer, Heidelberg, 2015.

References II

W. Penninckx, J. T. Mühlberg, B. J. Jan Smans, and F. Piessens.

Sound formal verification of Linux's USB BP keyboard driver.
In *NFM 2012*, vol. 7226 of *LNCS*, pp. 210–215, Heidelberg, 2012. Springer.

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 (SCP), 82:77–97, 2014.

R. Strackx, F. Piessens, and B. Preneel.

Efficient isolation of trusted subsystems in embedded systems.
In *Security and Privacy in Communication Networks*, vol. 50 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 344–361. Springer, 2010.

G. Vanspauwen and B. Jacobs.

Verifying protocol implementations by augmenting existing cryptographic libraries with specifications.
In *SEFM 2015*, vol. 9276 of *LNCS*, pp. 53–68. Springer, Heidelberg, 2015.

D. H. White and G. Lüttgen.

Identifying dynamic data structures by learning evolving patterns in memory.
In *TACAS 2013*, vol. 7795 of *LNCS*, pp. 354–369. Springer Berlin Heidelberg, 2013.