Noname manuscript No.
(will be inserted by the editor)

Joint Forces for Memory Safety Checking Revisited

Marek Chalupa · Jan Strejček · Martina Vitovská

the date of receipt and acceptance should be inserted later

Abstract We present an improved version of the memory safety verification approach implemented in Symbiotic 5, the winner of the *MemSafety* category at the *Competition on Software Verification* (SV-COMP) 2018. The approach can verify programs for standard errors in memory usage like invalid pointer dereference or memory leaking. It is based on instrumentation, static pointer analysis extended to consider memory deallocations, static program slicing, and symbolic execution. The improved version brings higher precision of the extended pointer analysis and further optimizations in instrumentation. It is implemented in the current version of Symbiotic, which contains also some improvements in program slicing and symbolic execution. We explain the approach in theory, describe implementation of selected components, and provide experimental results showing the impact of particular components.

1 Introduction

At SPIN 2018, we presented a successful approach [7] for checking *memory safety* of imperative programs. More precisely, the approach checks sequential (no multithreading or exceptions) imperative programs for the following types of errors:

invalid dereference – an operation reading from or writing to a byte in the memory that is not allocated (includes, for example, `null` pointer dereference and use-after-free),

invalid deallocation – an operation deallocating a memory block via a non-`null` pointer that does not point at the beginning of a memory block allocated on the heap (includes, for example, double free),

memory leak – a situation when a program returns from the main function without prior deallocation of all memory blocks allocated on the heap.

The approach combines static pointer analysis, instrumentation, static program slicing, and symbolic execution. It is implemented in Symbiotic 5 [8], a tool for

Masaryk University, Brno, Czech Republic
E-mail: {chalupa, strejcek, xvitovs1}@fi.muni.cz

verification of sequential C programs. With this approach, Symbiotic 5 won the *MemSafety* category of the recognized *Competition on Software Verification* (SV-COMP) 2018.

All techniques employed in the approach are well known and routinely used for verification of various program properties including memory safety. The particular combination of techniques is in the context of memory safety checking to our best knowledge original, although some similar combinations appeared before and we mention them later in related work. What we consider as the main contribution of this research is the specific way we use the results of pointer analysis to reduce instrumentation, which in turn enables program slicing to remove more program instructions than with use of basic instrumentation. Indeed, our experiments show that slicing applied after reduced instrumentation produces roughly half-sized programs compared to slicing with basic instrumentation. The smaller size of sliced programs brings significant increase in performance of the whole verification approach.

In this paper, we present an improved version of the approach, which includes in particular a more precise pointer analysis and a more efficient instrumentation. The changes are explicitly mentioned in the paper. The improved version of the approach is implemented in the current version of Symbiotic, which also brings slight improvements in slicing and symbolic execution. The experimental results presented in this paper are therefore completely new. While the conference paper [7] focuses mainly on theoretical description of the approach and experimental results, in this paper we also discuss more closely the implementation of individual components except the symbolic executor Klee [5], which is the only part of Symbiotic not written by us. Note that the code for instrumentation, static pointer analysis, and static program slicing is designed to be easily reusable.

In general, our verification approach combines a static data-flow analysis with compile-time instrumentation. Static data-flow analyses for memory safety checking [15, 19, 46] proved to be fast and efficient. However, they usually work with under- or over-approximation and thus tend to produce false alarms or miss some errors if they are applied as a stand-alone verification technique. Instrumentation, typically used for runtime monitoring, extends the program with code that tracks the memory allocated by the program and that checks correctness of memory accesses and absence of memory leaks. If a check fails, the instrumented program reaches an error location. Our approach combines a static data-flow analysis with instrumentation and static program slicing to get a small instrumented program that contains a reachable error location if and only if the original program contained a memory safety error. Finally, we run a reachability analysis on the instrumented and sliced program to reveal possible errors in memory manipulation contained in the original program. The last step is typically the most expensive part of our approach.

The basic workflow of our approach is shown in Figure 1. First, the program is instrumented. In principle, the inserted code tracks allocated memory and checks that memory accesses and deallocations touch only the allocated memory. With the help of a data-flow analysis, namely an extended form of pointer analysis, we optimize the instrumentation process to reduce the amount and complexity of inserted code. The optimizations can be divided into the following two classes

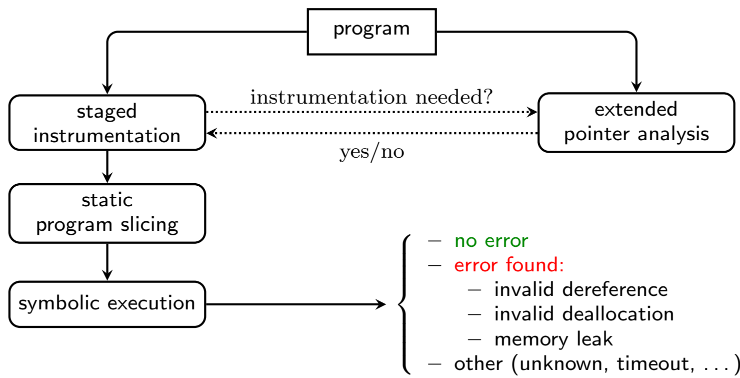

Fig. 1 The general workflow of our approach.

reflecting whether they reduce inserted checks (RC) or the code tracking allocated memory (RT):

(RC) These optimizations reduce the number and complexity of inserted checks. First of all, we do not insert a check before a pointer dereference if the pointer analysis guarantees that the operation is safe. For example, when the pointer analysis says that a given pointer always refers to the beginning of a global variable and a dereference via this pointer does not use more bytes than the size of the global variable, we know that the dereference is safe and we do not insert any check before it. Further, it may happen that the pointer analysis says that a pointer refers to some of the allocated memory blocks, but it cannot guarantee safety of the pointer dereference as it is not sure that the dereference is within bounds of the pointed block. In this case, we can sometimes use a simpler check than the general one. Finally, if the pointer analysis finds that a dereference is definitely invalid, we insert code jumping to an error location.

(RT) We reduce the code for tracking allocated memory such that it will track only information about the memory blocks that can be potentially used by some of the inserted checks.

These optimizations require a pointer analysis with slightly nonstandard properties. Since typical pointer analyses do not care whether a memory block was freed or its lifetime has ended, a standard pointer analysis could mark some parts of programs as safe even when they are not (e.g., dereferencing a freed memory). For this reason, we extend a pointer analysis such that it takes into account also instructions freeing heap-allocated memory and the lifetime of local variables. Due to (RT), we perform the instrumentation in two stages. During the first stage we insert checks and remember which memory blocks are relevant for these checks. The second stage inserts the code that tracks information about the relevant blocks.

In the next step, the instrumented program is statically sliced in order to remove the parts that are irrelevant for the reachability of inserted error locations. Finally, we use symbolic execution to perform the reachability analysis.

In our approach, we instrument the program with a real working code instead of inserting calls to place-holder functions interpreted by a verifier tool. In this way, the program is extended in a tool-independent manner and any tool working with the same program representation can be used to perform the reachability analysis.

Moreover, the instrumented program can be even compiled and run (provided the original program was executable). The disadvantage is that the reachability analysis tools that have problems with precise handling of complicated heap-allocated data structures may struggle with handling the inserted functions since these typically use structures like linked lists or search trees for tracking the state of allocated memory blocks.

The presented verification approach is implemented in the tool Symbiotic, which builds upon the llvm framework [28,30]. Hence, each analyzed C program is compiled into llvm before the instrumentation starts. llvm is an intermediate representation language on the level of instructions that is suitable for verification for its simplicity. Examples contained in this paper are also in llvm, which is slightly simplified to improve readability. For the needs of presentation, we explain a few of the llvm instructions:

- `alloca` instruction allocates memory of the given size on the stack and returns its address,
- `load` reads a value from the address given as its operand,
- `store` writes a given value to the memory on the address given as the second operand,
- `call` instruction is used to call a given function. Function `malloc` allocates memory of the given size on the heap and returns its address. Function `free` deallocates memory on the given address allocated on the heap.

When there is any other instruction used in the paper, its semantics is described at a relevant place in the text.

In its theoretical part, the paper focuses mainly on the instrumentation and the extension of pointer analysis, as we use a standard static program slicing based on dependency graphs [17] and a standard symbolic execution [26] implemented in Klee [5]. More precisely, Section 2 describes the basic version of code instrumentation for checking memory safety that does not use any pointer analysis. Section 3 then introduces the extended pointer analysis and explains the instrumentation optimizations (RC) and (RT). The implementation of individual components employed by the approach (except the symbolic executor Klee [5]) is discussed in Section 4. In particular, we introduce a tool for configurable instrumentation of llvm bitcode supporting instrumentation in stages, a library providing several different pointer analyses of llvm bitcode, and an llvm bitcode slicer. Section 5 presents experimental results comparing Symbiotic with state-of-the-art tools for memory safety checking and illustrating the contribution of instrumentation optimizations and program slicing to the overall performance. Section 6 summarizes advantages and disadvantages of our approach. Related work is discussed in Section 7.

2 Basic Instrumentation

The basic version of instrumentation inserts code that tracks all allocated memory blocks (including global and stack variables) and checks correctness of all memory accesses and deallocations just before their execution. Similarly as Jones and Kelly [25], for every allocated block of memory we maintain a record with its address and size. The records are stored in three linked lists:

– *HeapList* for blocks allocated on the heap,
– *StackList* for blocks allocated on the stack,
– *GlobalsList* for global variables.

Additionally, we maintain *DeallocatedList* for blocks on the heap that were already deallocated. This list can be safely omitted as it serves only to provide better error descriptions. More precisely, the information in this list enables us to distinguish double free from generic invalid deallocation, or use-after-free from vague invalid dereference error. In the following, we focus on the core functionality of inserted code and thus our presentation does not mention the maintenance and utilization of *DeallocatedList*.

To maintain the lists that track the state of the memory, we call the function `remember_heap(`*addr*, *size*`)` or `remember_stack(`*addr*, *size*`)` after each memory allocation on the heap or stack, respectively. Also, we call `remember_global(`*addr*, *size*`)` at the beginning of the `main` function for each global variable. As recently allocated blocks tend to be accessed more often than the older blocks, we add new records to the beginning of the lists. Before every deallocation, we call function `handle_free(`*addr*`)` that checks that *addr* is either `null` or it refers to the beginning of a memory block allocated on the heap. If the answer is positive, it removes the corresponding record from *HeapList*. If the check fails, we jump to an error location and report invalid deallocation. Allocations on the stack are destroyed when the corresponding function finishes. To reflect this behavior, we use functions `fun_entry()` and `fun_exit()`. Whenever we enter a function, we call `fun_entry()` that adds a mark at the beginning of *StackList*. Before returning from a function, we call `fun_exit()` that removes the latest mark and all the records added to *StackList* after this mark. Further, before every instruction loading or storing n bytes from/to the address *addr* we call function `check_pointer(`*addr*, n`)` to check that the memory operation is safe. This function goes through the lists and looks for a record of the memory block containing the accessed n bytes. If there is no such record (which includes the case when some record contains only part of the n accessed bytes), we jump to an error location and report invalid dereference. Finally, we insert `check_leaks()` before each return from the `main` function to check that *HeapList* is empty. If the list contains some record, we jump to an error location and report the memory leak corresponding to the record.

In contrast to the instrumentation presented in the conference paper [7], the current version also supports allocations on the stack that are local to a scope other than the scope of a function. In LLVM, such explicit scope of memory is delimited by the functions `llvm.lifetime.start` and `llvm.lifetime.end` with arguments pointing to the relevant memory. Instead of allocating more memory blocks with non-overlapping lifetimes, LLVM can allocate one memory block and use `llvm.lifetime.start` and `llvm.lifetime.end` repeatedly on this block. Hence, after every call of `llvm.lifetime.end` we remove the corresponding record from the *StackList* by calling function `remove_stack`. Further, after every `llvm.lifetime.start` we call `remember_stack` that checks whether *StackList* contains the corresponding record and if not, it adds it.

During runtime, there can be situations when a pointer is incorrectly shifted to a different valid object in memory (e.g., when two arrays are allocated on the stack one next to the other, a pointer may overflow from the first one to the second one). In this case, the checking function finds a record for the object pointed to by the

```
1. %p = alloca i32*                          %p = alloca i32*
   call remember_stack(%p, 8)                call remember_stack(%p, 8)
   call check_pointer(%p, 8)
2. store null to %p                          store null to %p
3. %addr = call malloc(20)                   %addr = call malloc(20)
   call remember_heap(%addr, 20)             call remember_heap(%addr, 20)
   call check_pointer(%p, 8)
4. store %addr to %p                         store %addr to %p
   call handle_free(%addr)                   call handle_free(%addr)
5. call free(%addr);                         call free(%addr);
   call check_pointer(%p, 8)
6. %tmp = load %p                            %tmp = load %p
   call check_pointer(%tmp, 4)               call check_fail()
7. store i32 1 to %tmp                       store i32 1 to %tmp
```

Fig. 2 Instrumentation of a code with an invalid pointer dereference on line 7. The code on the left is instrumented by the basic instrumentation while the code on the right is instrumented using the optimizations (RC) described in Section 3. We assume that the width of a pointer is 8 bytes and the width of an integer (in LLVM denoted as the type `i32`) is 4 bytes.

pointer and it does not raise any error even though the pointer points outside of its base object. To overcome this problem, some approaches instrument also every pointer arithmetic operation [13,25,41]. We do not instrument pointer arithmetic operations as we do not execute the code but pass it to a verification tool that keeps strict distinction between objects in memory. Therefore, a pointer derived from an object cannot overflow to a different object.

An example of a basic instrumentation is provided in Figure 2 (left). Allocations on lines 1 and 3 are instrumented with calls to `remember_stack` and `remember_heap`, respectively. The variable `%addr` keeps the address of the memory allocated by the call to `malloc` on line 3. Subsequently, this address is stored to the memory pointed to by `%p`. The memory pointed to by `%addr` is then freed on line 5 and `handle_free` is called before this event. Function `check_pointer` is called before each `load` and `store` instruction. The call of `check_pointer` before line 7 reveals use-after-free error as the value loaded from the address `%p` on line 6 is the address of the memory allocated on line 3 and freed on line 5.

The presented basic instrumentation correctly transforms real memory safety errors into reachable error locations in the following sense.

Theorem 1 *A given program has a run containing an invalid dereference error if and only if the program after basic instrumentation has a run reaching an error location and reporting the invalid dereference error. The same holds for invalid deallocations and memory leaks.*

Proof (Sketch) The statements for invalid dereferences and deallocations follow from several facts. First, instrumentation does not change the order in which the original program instructions are executed. Moreover, the inserted code does not modify the data manipulated by the original instructions. Hence, every run of the original program exactly corresponds to a run of the instrumented program (which in addition executes some inserted code), and vice versa. However, the inserted code can stop a run by jumping to an error location if some inserted check fails.

Second, the information stored in *HeapList*, *StackList*, and *GlobalsList* is valid in the sense that it represents exactly the memory blocks that are currently allocated.

More precisely, if some instruction of the original program creates or destroys a memory block, this change is reflected in the lists before the next original instruction is executed.

Finally, the basic instrumentation inserts a check in front of every original instruction performing a pointer dereference or a deallocation. As these checks work with valid information in the lists and check exactly the correctness of the upcoming dereference or deallocation, a run jumps to an error location and reports the invalid dereference or deallocation if and only if the upcoming pointer dereference or deallocation would be invalid.

The statement for memory leaks also comes from the fact that the information in the lists is valid, the check for memory leaks is called at every return from the `main` function, and it checks whether some block allocated on the heap is not explicitly deallocated, which is exactly memory leaking. □

The main disadvantage of the basic instrumentation is that it tracks all memory allocations and instruments all dereferences and deallocations. The amount of inserted function calls is therefore usually very large. As these calls use variables of the original program as arguments, many instructions of the original code can have a potential effect on the reachability of inserted error locations and thus cannot be removed by slicing.

3 Optimized Instrumentation

All suggested instrumentation optimizations rely on an extended pointer analysis. Hence, we first recall the standard pointer analysis and describe its extension.

3.1 Extended Pointer Analysis

Roughly speaking, a standard pointer analysis computes a *points-to set* for each pointer variable. A points-to set of a pointer variable contains all *memory locations* to which the variable may point to. Here, a memory location is an abstraction of a concrete object located in memory during runtime. A frequent choice used also by our analysis is to abstract these objects by instructions that allocated them. For example, the object allocated on line 3 in Figure 2 is represented by the memory location `3:malloc(20)` reflecting the line number and the allocation function. Note that one memory location can represent several objects in the case that the program can execute the allocation instruction multiple times. This can happen, for example, when the allocation is within a program loop or in a recursive function. Besides memory locations, points-to sets can also contain two special elements: `null` if the pointer's value may be `null`, and `unknown` if the analysis fails to establish information about some referenced memory location.

The precision of pointer analysis can be tuned in several directions. A pointer analysis is called *flow-sensitive* [21] if it takes into consideration the flow of data in the program and computes specific points-to information for every control location in the program. On the contrary, *flow-insensitive* analyses ignore the execution order of instructions and compute summary information about pointers that holds at any control location in the program. For instance, in Figure 2 a flow-insensitive

analysis would tell us that `%tmp` may point either to `null` or to the memory location `3:malloc(20)` due to the assignments on lines 2 and 4. The flow-sensitive analysis can tell us that `%tmp` may point only to `3:malloc(20)`. In the context of standard programming languages, one has to specify a control location when asking a flow-sensitive pointer analysis for the points-to set of some pointer variable. When working with LLVM, we do not do that as LLVM maintains variables (also called *register* in this context) in SSA form [12] where each variable is set by a single instruction only. When we refer to a points-to set of a pointer variable, we thus mean the points-to set immediately after the instruction setting the variable. A pointer analysis is called *field-sensitive* if it differentiates between individual elements of arrays and structures. We achieve field-sensitivity by refining information in points-to sets with offsets (e.g., a pointer variable p points to memory location A at offset 4).

Standard pointer analyses ignore information whether a memory block was freed or whether the lifetime of a local variable has ended because of the end of its scope. Even though such events do not change pointer values, they are crucial if we want to use pointer analysis to optimize the instrumentation process. Consider the dereference on line 7 in Figure 2. Usual flow- and field-sensitive pointer analysis tells us that the pointer `%tmp` points to the location `3:malloc(20)` at offset 0 and thus writing 4 bytes to that memory seems to be safe. However, it is not as this memory has been already freed on line 5.

There exist sophisticated forms of pointer analysis that can model the heap and the stack and provide information about deallocation and ceased lifetime of memory objects (e.g., shape analysis [21,38,16]), but these are relatively expensive for our use case. Instead, we extended a simple flow- and field-sensitive inclusion-based [1,22] pointer analysis so that it can track whether a pointer variable can possibly point to an invalidated memory (i.e., a memory that was explicitly freed or its lifetime ended).

For every pointer variable p, the extended pointer analysis computes a points-to set

$$\mathit{ptset}(p) \subseteq (\mathit{Mem} \times \mathit{Offset}) \cup \{\texttt{null}, \texttt{unknown}, \texttt{invalidated}\},$$

where Mem is the set of memory locations, $\mathit{Offset} = \mathbb{N}_0 \cup \{?\}$ is the set of non-negative integers extended with the special element '?' denoting an unknown offset, and `invalidated` is the new special memory location representing memory that has been deallocated or destroyed as its lifetime ended. We also assume that $\mathit{ptset}(p) \neq \emptyset$ for any pointer variable p. Should it be the case, we set $\mathit{ptset}(p) = \{\texttt{unknown}\}$.

To compute points-to sets for pointer variables, our analysis actually computes points-to set $\mathit{ptset}(M, C)$ for each memory location M storing a pointer and each control location C of the program. The computation of these points-to sets proceeds as a standard data-flow analysis: $\mathit{ptset}(M, C)$ is derived from sets $\mathit{ptset}(M, C')$ for control locations C' immediately preceding the control location C and from the effect of the instruction corresponding to C. Computation of all these points-to sets at all control locations repeats until a fixpoint is reached.

Pointer analyses often use only *weak updates*, which means that $\mathit{ptset}(M, C)$ is set to be the union of all $\mathit{ptset}(M, C')$ for C' preceding C enlarged according to the effect of the instruction associated with C. The rationale for weak update is that M can represent more than one memory object during runtime and the analysis does not know which of the objects is changed by the current instruction. However, if an

instruction changes a memory location that represents a single memory object, we can apply *strong update* which computes the points-to set just from the effect of the instruction (i.e., strong update overrides the information from preceding control locations). We say that a memory location is *single-instance* if it never represents two or more allocated memory objects at the same time. In particular, every allocation function or instruction that is executed at most once during every run corresponds to a single-instance memory location. Our extended pointer analysis aims to identify single-instance memory locations and applies strong updates when possible.

For the measurements presented in the conference paper [7], we used an extended pointer analysis that applies strong update only for pointer assignments. Now we apply strong update in two additional cases.

a) When a single-instance memory location representing an object on the stack is destroyed because of the end of its scope, we replace this memory location by `invalidated` in all points-to sets associated to the current control location. Note that we cannot do this when the memory location is not single-instance (e.g., when it represents some local variable of a recursive function) as only the newest object represented by the memory location is destroyed. In this case, we apply weak update that adds `invalidated` to all points-to sets containing the memory object.
b) If the function `free` is applied to a pointer whose points-to set contains a single-instance memory location as the only element, we know that the object represented by this memory location is deallocated. Thus, we replace the memory location by `invalidated` in all points-to sets associated to the current control location.

The new applications of strong update improve the precision of the extended pointer analysis. For example, consider the call `free(%addr)` on line 5 of Figure 2. As the points-to set of `%addr` contains only the single-instance memory location `3:malloc(20)`, we replace this memory location by `invalidated` in all points-to sets. In particular, the points-to set of `1:alloca i32*` contains just `invalidated` after this deallocation, while it would contain both `invalidated` and `3:malloc(20)` if we apply weak update only. Thanks to the improved precision, the extended pointer analysis immediately implies that the dereference on line 7 performs an invalid dereference as `%tmp` points to invalidated memory.

3.2 Reduction of Checks (RC)

The function `check_pointer`$(addr, n)$ used by our instrumentation approach to check validity of memory accesses is not cheap. It searches the lists of records (*StackList*, *HeapList*, and *GlobalsList*) for the one that represents the memory block where *addr* points to. Hence, it has linear complexity with respect to the number of records in the lists. Here we present improvements that can completely omit some unnecessary checks and replace some of the remaining checks with simpler ones.

The extended pointer analysis can often guarantee that each possible memory dereference performed by a particular instruction is safe. Let us assume that an instruction reads or writes n bytes from/to the memory referenced by a pointer

```
1. %array = alloca [10 x i32]
   call remember_stack(%array, 10*4)
2. %m = call input()
3. %tmp = getelementptr %array, %m
   call check_bounds(%tmp, 4, %array, 0, 40, 0, 40)
4. store 1 to %tmp
```

Fig. 3 A code instrumented with `check_bounds`. Recall the assumption that the width of an integer (`i32`) is 4 bytes.

variable p. The extended pointer analysis guarantees its safety if $ptset(p)$ contains neither `null` nor `unknown` nor `invalidated`, and for every $(A, \mathit{offset}) \in ptset(p)$ it holds that every object represented by the memory location A contains at least $\mathit{offset} + n$ bytes. Formally, we know that the access is safe if

- $ptset(p) \cap \{\texttt{unknown}, \texttt{null}, \texttt{invalidated}\} = \emptyset$ and
- for each $(A, \mathit{offset}) \in ptset(p)$ it holds that $\mathit{offset} \neq ?$ and $\mathit{offset} + n \leq size(A)$,

where $size(A)$ denotes the size of the memory objects represented by A if it is known at compile time, otherwise it denotes 0 (and thus the condition does not hold as $n \geq 1$). Hence, before instrumenting a memory access with a check, we query the extended pointer analysis. If the analysis says that the memory access is safe, the check is not inserted. For example, in Figure 2 the dereferences of the variable `%p` on lines 2, 4, and 6 are safe and thus need not be instrumented with any check.

If the extended pointer analysis does not guarantee safety of a memory access, we need to instrument the access. However, in many cases, we can call some of the following functions which are cheaper than the generic `check_pointer` function:

- `check_fail`
- `check_bounds`
- `check_heap`
- `check_stack`
- `check_globals`

Now we describe the semantics of these functions and situations when they replace the generic check.

We start with the function `check_fail`. The extended pointer analysis may also detect that a memory access via a pointer variable p has to be invalid as $ptset(p) \subseteq \{\texttt{invalidated}, \texttt{null}\}$. Note that the instruction performing an invalid access may be unreachable and thus we do not report the invalid dereference immediately. Instead of calling the generic check, we insert a call to `check_fail()` that jumps to an error location. For example, this happens for the dereference on line 7 in Figure 2, where the pointer analysis tells us that `%tmp` may point only to invalidated memory.

The function `check_bounds` is more involved. Let us again assume that there is an instruction accessing n bytes in the memory via a pointer variable p_1 and such that the extended pointer analysis cannot guarantee its safety. Further, assume that the value of p_1 has been computed as a pointer p_0 shifted by some number of bytes. Figure 3 provides a simple code where such situation arises. In this example, an array of ten integers is allocated on line 1. The function `input()`

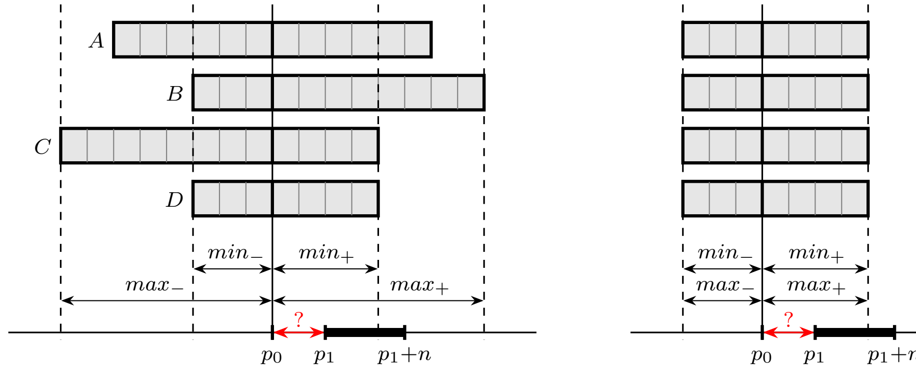

Fig. 4 The figure on the left depicts the case that $\mathit{ptset}(p_0) = \{(A,6),(B,3),(C,8),(D,3)\}$ and the sizes of memory locations A, B, C, D are $12, 11, 12, 7$, respectively. Assume that the pointer analysis cannot guarantee safety of an instruction accessing n bytes at the address given by p_1, but we know that p_1 is derived from p_0. Instead of `check_pointer`(p_1, n), we can insert `check_bounds`$(p_1, n, p_0, min_-, min_+, max_-, max_+)$. The figure on the right depicts the special case where $min_- = max_-$ and $min_+ = max_+$.

called on line 2 reads user input. The instruction `%tmp = getelementptr %array, %m` on line 3 returns the address of the m-th element of the array, i.e., the address `%array` increased by $4m$ bytes. Line 4 stores integer 1 on this address. The extended pointer analysis cannot determine the offset of this address as it depends on the user input, and thus a check needs to be called before line 4 is executed. However, in situations like this we may be able to insert a call to the simpler `check_bounds` function instead of the complex `check_pointer`.

We use the function `check_bounds` instead of the generic check if p_0 can point only to memory blocks of sizes known at compile time and all potential offsets of p_0 are also known. Formally, we insert an optimized check before a potentially unsafe dereference of p_1 if

- $\mathit{ptset}(p_0) \cap \{\texttt{unknown}, \texttt{null}, \texttt{invalidated}\} = \emptyset$ and
- for each $(A, \mathit{offset}) \in \mathit{ptset}(p_0)$ it holds that $\mathit{offset} \neq\ ?$ and $\mathit{size}(A)$ is known at compile time.

Under these conditions, we can easily compute lower and upper bounds on the number of bytes allocated to the left and to the right of p_0 as illustrated by the example in Figure 4 (left). The bounds are computed as follows:

$$\begin{aligned}
min_- &= \min\{\mathit{offset} \mid (A, \mathit{offset}) \in \mathit{ptset}(p_0)\}\\
max_- &= \max\{\mathit{offset} \mid (A, \mathit{offset}) \in \mathit{ptset}(p_0)\}\\
min_+ &= \min\{\mathit{size}(A) - \mathit{offset} \mid (A, \mathit{offset}) \in \mathit{ptset}(p_0)\}\\
max_+ &= \max\{\mathit{size}(A) - \mathit{offset} \mid (A, \mathit{offset}) \in \mathit{ptset}(p_0)\}
\end{aligned}$$

Now we can insert the call of `check_bounds`$(p_1, n, p_0, min_-, min_+, max_-, max_+)$ to check validity of the memory access to n bytes pointed by p_1. The function computes the difference $o = p_1 - p_0$ and checks whether the access is

- within the lower bounds, i.e., $-min_- \leq o$ and $o + n \leq min_+$, and thus safe, or
- exceeds the upper bounds, i.e., $o < -max_-$ or $o + n > max_+$, and thus invalid.

```
1. %n = call input()
2. %array = call malloc(%n)
   call remember_heap(%array,%n)
3. %tmp = getelementptr %array, 10
   call check_heap(%tmp, 4)
4. store 1 to %tmp
```

Fig. 5 A code instrumented with `check_heap` which searches only *HeapList*.

If none of the two checks succeeds, the generic `check_pointer(`p_1, n`)` is called. Notice that when $min_- = max_-$ and $min_+ = max_+$, which is the case depicted in Figure 4 (right), one of the two checks against bounds always succeeds and the generic check is never called. This is also the case of Figure 3, where pointer analysis determines that `%array` points to the beginning (i.e., at offset 0) of the block of size 40. Therefore $min_- = max_- = 0$ and $min_+ = max_+ = 40$ and the call to `check_bounds(%tmp, 4, %array, 0, 40, 0, 40)` is inserted.

The function `check_heap` is inserted when `check_fail` or `check_bounds` are not applicable, but the extended pointer analysis guarantees that the dereferenced pointer p points to the heap, i.e.,

- `unknown` $\notin ptset(p)$ and
- for each $(A, \mathit{offset}) \in ptset(p)$, A represents memory allocated on the heap.

The function `check_heap` directly searches just *HeapList* instead of searching all the lists. A typical application is shown in Figure 5. Line 2 allocates an array of n bytes on the heap. The instruction `%tmp = getelementptr %array, 10` on line 3 returns the address of the 10th element of the array. Line 4 stores integer 1 on this address. As the extended pointer analysis cannot determine the size of `%array`, `check_bounds` cannot be used. However, the analysis has the information that we are dereferencing a pointer that points to the heap. Therefore, a call to `check_heap` is inserted instead of the usual `check_pointer`.

Analogously, we use the function `check_stack` or `check_globals` if the pointer analysis implies that p points to stack or to some global variable, respectively.

Note that the original version of our approach [7] uses `check_bounds` as the only simpler check. Moreover, the original approach actually uses a simpler version of `check_bounds` applicable only if $min_- = max_-$ and $min_+ = max_+$, which is the situation depicted in Figure 4 (right).

3.3 Reduction of Memory Tracking Code (RT)

Although the previous optimizations simplify or eliminate checks of dereference safety, the approach still tracks all memory blocks. However, it is sufficient to track only memory blocks that are relevant for some check. For example, the code in Figure 2 (right) remembers records for both allocations on lines 1 and 3, but no record corresponding to the allocation on line 1 is ever used: `handle_free(%addr)` searches only *HeapList* and the extended pointer analysis tells us that the dereference of `%tmp` on line 7 is invalid. Hence, the call to `remember_stack` inserted after line 1 can be safely omitted.

In general, we always track all blocks allocated on the heap as they are relevant for checking memory leaks. Further, we track all memory blocks if the points-to set of some dereferenced pointer contains the element `unknown` meaning that the pointer can point anywhere. Otherwise, we do not track global variables and memory objects allocated on stack that are not relevant for any inserted check. An object is relevant for `check_pointer(`$addr, n$`)` if $ptset(addr)$ contains a memory location corresponding to the object. The same principle applies to `check_stack` and `check_globals`. Recall that `check_heap` searches only blocks allocated on the heap and these are all tracked anyway. The situation for `check_bounds(`$p_1, n, p_0, min_-, min_+, max_-, max_+$`)` is more interesting as we do not need to track all blocks in $ptset(p_1)$. We can safely ignore the blocks corresponding to a memory location A if $(A, offset) \in ptset(p_0)$ holds only for one *offset* and this *offset* satisfies $offset = min_-$ and $size(A) - offset = min_+$. Indeed, checking validity of the dereference of p_1 against the tracked information would have exactly the same effect as the check against the lower bounds min_- and min_+, which is done as the first step of `check_bounds`. For example, the blocks corresponding to the memory location D in Figure 4 (left) are not relevant for the `check_bounds` exactly for this reason. The same holds for all blocks in Figure 4 (right).

In fact, our instrumentation process has two stages.

1. In the first stage, checks are inserted as described before. Additionally, for every inserted call to `check_pointer`, `check_stack`, and `check_globals` we remember the memory locations contained in the points-to set of their first argument, which is the pointer variable being dereferenced. We also remember `unknown` if it is contained in the points-to set. For every inserted call to `check_bounds(`$p_1, n, p_0, min_-, min_+, max_-, max_+$`)`, we remember all memory locations A such that $(A, offset) \in ptset(p_1)$ and $offset > min_-$ or $size(A) - offset > min_+$. In the first stage, we also insert all calls to `remember_heap`, `handle_free`, `fun_entry`, `fun_exit`, and `remove_stack`.
2. The second stage inserts calls to `remember_global` and `remember_stack` if the memory location representing the allocated global variable or block on stack has been remembered in the first stage or if `unknown` has been remembered in the first stage. Further, we insert the call to `check_leaks` at the end of `main` function only if some call to `remember_heap` was inserted in the first stage.

In Figures 2 (right) and 3, the instrumentation with (RT) optimization would not insert any call to `remember_stack`.

3.4 Correctness

The instrumentation optimized either with the (RC) reduction or both (RC) and (RT) reductions still correctly transforms memory safety errors to reachable error locations assuming that it gets valid points-to sets. A points-to set of a pointer variable p is *valid* if it satisfies the following conditions during all runs of the program:

– If p was derived from the address of a (still) allocated block, the $ptset(p)$ contains $(A, offset)$ or $(A, ?)$ or `unknown`, where A is the memory location representing the block and *offset* is the relative offset to the base address of the allocated block.

– If p does point into a block of deallocated memory or memory that was never allocated, then $ptset(p)$ contains `invalidated` or `unknown`.
– If p has the value `null`, then $ptset(p)$ contains `null` or `unknown`.
– If the value of p was loaded from previously uninitialized memory, then $ptset(p)$ contains `unknown`.

We decided to put validity of points-to sets as a precondition rather than proving it because we describe pointer analyses only on an intutive level.

Theorem 2 *Assume that we are given a program and a valid* $ptset(p)$ *for each pointer variable of the program. The program has a run containing an invalid dereference error if and only if the program after instrumentation optimized with (RC) reduction has a run reaching an error location and reporting the invalid dereference error. The same holds for invalid deallocations and memory leaks.*

Proof (Sketch) The instrumentation optimized with (RC) reduction behaves like the basic instrumentation with the difference that it omits some checks and uses simpler checks before some dereferences. However, a check is omitted only before a dereference whose validity is guaranteed by the (valid) point-to set of the dereferenced variable. Further, every inserted simpler check has the same effect as the generic `check_pointer` if the relevant points-to set is valid.

The statements for deallocations and memory leaks follow directly from Theorem 1 as (RC) reduction changes only instrumentation of dereferences. □

Theorem 3 *Assume that we are given a program and a valid* $ptset(p)$ *for each pointer variable of the program. The program has a run containing an invalid dereference error if and only if the program after instrumentation optimized with (RC) and (RT) reductions has a run reaching an error location and reporting the invalid dereference error. The same holds for invalid deallocations and memory leaks.*

Proof (Sketch) The checks of dereference validity are inserted in the same way as by the instrumentation optimized with (RC) reduction. The change introduced by (RT) reduction is that we do not track global variables and memory blocks allocated on the stack that are not relevant for any of these checks according to the corresponding points-to sets. The statement for invalid dereferences thus follows from Theorem 2 and the assumption that points-to sets are valid.

Regarding deallocations and memory leaks, the only relevant change introduced by (RT) reduction is that we do not insert `chect_leaks` to programs that do not allocate memory on the heap. Hence, validity of the statements for deallocations and memory leaks follows from Theorem 2. □

In general, inserting fewer calls to functions that create records has a positive effect on the speed of reachability analysis since *StackList* and *GlobalsList* are shorter. All the described extensions together can significantly reduce the amount of inserted code. This has also a positive effect on the portion of code possibly removed by static program slicing before the reachability analysis.

4 Implementation

The described approach is implemented in Symbiotic tool, revision tag `sttt`[1]. The tool consists of three main parts, namely *instrumentation tool*, *slicer*, and the

[1] `https://github.com/staticafi/symbiotic/releases/tag/sttt`

external state-of-the-art open-source symbolic executor KLEE [5] licensed under the University of Illinois license. The instrumentation and slicing modules rely on our library called `dg` that provides dependence graph construction and various pointer analyses including the extended pointer analysis described in Section 3.

The C program to be verified is first translated to LLVM [30] using CLANG [11]. The translated program is then instrumented by the instrumentation tool and optimized with selected optimizations provided by the LLVM framework. Further, the program is sliced and the LLVM optimizations are applied again. Finally, KLEE is executed on the sliced and optimized code to check reachability of the inserted error locations.

All parts of SYMBIOTIC except KLEE are licensed under the MIT open-source license and can be reached via:

https://github.com/staticafi/symbiotic

Now we describe the `dg` library, the instrumentation tool, and the slicer in more details. All these components are implemented in C++.

4.1 The `dg` Library

The `dg` library incorporates various algorithms for program analysis and dependence graphs building [17]. The library includes configurable pointer analyses and precise reaching definitions analyses written in a generic way and instantiated for LLVM bitcode. The extended pointer analysis described in Section 3.1 is also a part of the `dg` library.

Similar to Hind et al. [22], pointer analyses in `dg` use *sparse evaluation graphs* when computing information about pointers. Intuitively, sparse evaluation graph is a subgraph of the control flow graph that contains only nodes relevant to pointer analysis and edges representing paths between these nodes. The pointer analysis builds the sparse evaluation graph for each function of the analyzed program and connects these graphs to one *interprocedural sparse evaluation graph* (ISEG) by adding edges from call-sites to entries of the called functions and from exits of functions to the corresponding return-sites. As a side effect, recursive calls or repeated calls of the same function get transformed into loops in ISEG. Note that a memory location must be single-instance if the corresponding allocation instruction does not belong to any program's loop and is not in a function that can be called repeatedly. Hence, each allocation node that is not on a cycle in ISEG has to represent a single-instance memory location.

4.2 Instrumentation Tool

Instead of implementing a single-purpose instrumentation for memory safety checking, we developed a configurable instrumentation tool called `sbt-instrumentation` [45]. The basic schema of `sbt-instrumentation` is depicted in Figure 6. Besides the LLVM bitcode to be instrumented, the tool needs to be supplied with two files created by a user: a file with definitions of so called *instrumentation functions* whose calls will be inserted into the code, and a JSON

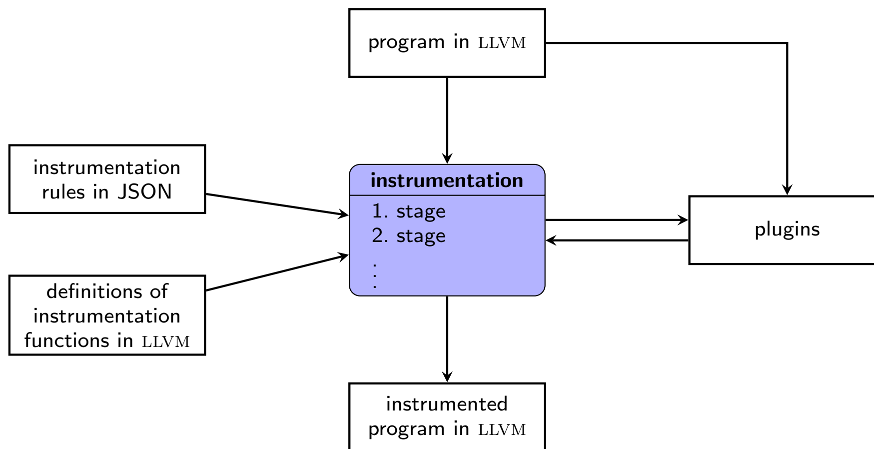

Fig. 6 The schema of the configurable instrumentation tool `sbt-instrumentation`.

file with *instrumentation rules* that define how the LLVM bitcode should be instrumented with calls of instrumentation functions. In practice, the fact that our tool can insert just calls to instrumentation functions is not a restriction as these functions can contain arbitrary code.

An instrumentation rule consists of two parts saying *when* it should be applied and *what* its effect is. The first part of an instrumentation rule is specified by

- functions in which the rule is applied (typically `main` or all functions),
- a sequence of instructions that should be matched, and
- conditions under which the rule is applied.

The second part describes

- the instrumentation function call that should be inserted,
- where it should be inserted (before or after the matched sequence), and
- information-gathering effects of the rule, namely setting flags and remembering values or variables used by the matched instructions in an auxiliary list.

Moreover, there are two other kinds of rules, namely rules for instrumentation of global variable declarations and rules instrumenting entry and exit points of functions.

Each rule can be guarded by conditions of several kinds. A condition can claim that

- a given flag has a particular value,
- a given value or a variable has been remembered earlier (or not),
- an external plugin returns a particular answer on a given query constructed with parts of matched instructions.

A rule with conditions is applied only if all conditions are satisfied. For example, in memory safety checking we use the extended pointer analysis as a plugin in order to instrument only dereferences that are not safe and to insert simpler checks if possible.

The instrumentation proceeds in one or more stages, each stage defined by a set of instrumentation rules. In each stage, the tool goes through all instructions of the

```
{
    "findInstructions": [
    {
        "returnValue": "*",
        "instruction": "load",
        "operands": ["<op>"]
    }
    ],
    "newInstruction": {
        "returnValue": "*",
        "instruction": "call",
        "operands": ["check_fail"]
    },
    "where": "before",
    "conditions": [{"query":["isInvalid", "<op>"], "expectedResults":["true"]}],
    "in": "*"
},
```

Fig. 7 Example of an instrumentation rule that inserts a call to `check_fail` before every `load` instruction, given that the load is definitely invalid.

given LLVM bitcode and it looks for instructions matching any instrumentation rule of the current stage. If a match is found, conditions of the instrumentation rule are evaluated. This is where plugins can be queried. If conditions are satisfied, the rule is applied, i.e., a new code is inserted according to the rule and some information can be gathered for a later use. An example of an instrumentation process that gathers such information and uses it in the next stage is given in Section 3.3. Indeed, the (RT) reduction is enabled by the information about the inserted checks that is gathered in the first stage of the optimized instrumentation.

We give an example of an instrumentation rule in Figure 7. This rule instructs the instrumentation to insert a call to the function `check_fail` before any load that is definitely invalid. In details, the rule comprises several parts. The `findInstructions` field defines a sequence of instructions to be instrumented. In this case, we are looking for sequences of the length one consisting of a single `load` instruction. When a `load` instruction is found, its operand is denoted by the variable `<op>` (and the loaded value is ignored). The `newInstruction` field defines the new instruction that will be inserted if the given sequence is matched. Here a call to the function `check_fail` with no arguments will be constructed. As the only operand of the call instruction is the function itself, the call will have no arguments, i.e., the new instruction is going to be `call check_fail()`. The `where` field determines that the new instruction will be inserted `before` every matched `load` instruction and the `in` field states that the rule will be applied in every function (thus the `*` sign). However, the rule takes effect only if the condition given by the `conditions` field is satisfied. Hence the call will be inserted only if an external plugin answers `true` to the query `isInvalid`. In this case, the returned answer is `true` only if the points-to set of `<op>` is a subset of {`null`, `invalidated`}.

After the last instrumentation stage, the instrumented program is linked with the instrumentation functions. The result of the instrumentation process is again an LLVM bitcode.

The `sbt-instrumentation` tool is distributed as a part of SYMBIOTIC and comes with two predefined configurations used for program analysis, namely a configura-

tion for checking memory safety as described in Section 3, and a configuration for checking signed integer overflows. The latter configuration inserts a check before every binary operation over signed integers that may potentially overflow. The decision whether an operation may overflow is based on the results of a range analysis [40].

The `sbt-instrumentation` tool together with the predefined configurations for checking memory safety and integer overflows can be found at:

https://github.com/staticafi/sbt-instrumentation

It uses an open-source parser for the JSON format JsonCpp[2]. The repository also contains scripts for downloading libraries that are necessary for plugins used by the predefined configurations.

4.3 Slicer

Since we have not found any suitable program slicer for LLVM bitcode, we created a tool called `sbt-slicer` on top of the `dg` library. Instead of using the traditional two-pass algorithm introduced in Horwitz et al. [24], we use a variant of the basic slicing algorithm based on dependence graphs [17] extended for inter-procedural slicing. In this algorithm, dependence graphs for procedures are connected with inter-procedural edges and the slice is obtained by one backward search.

When computing the dependence graph of the program to be sliced, data dependencies are derived from results of byte-precise reaching definition analysis. This analysis follows the classical data-flow approach and uses information about pointers provided by a field-sensitive and flow-insensitive inclusion-based pointer analysis [1]. The slicer can be also configured to use flow-sensitive pointer analysis, but the computation is more expensive and, according to our experience, it does not bring any positive effect on performance of Symbiotic.

Control dependencies are computed using the algorithm by Ferrante et al. [17]. This traditional algorithm assumes that every program path terminates, which may lead to incorrect slices in the presence of non-terminating loops: a non-terminating loop may be sliced away and hence a previously unreachable code may become reachable. Although we have not experienced this problem in our experiments, it may lead to reporting false alarms. We currently work on the implementation of a termination-sensitive control dependence algorithm.

By default, Symbiotic considers slicing criteria to be the calls to the function `__assert_fail` that comes from the expansion of the standard macro `assert`, and the function `__VERIFIER_error` that is the official marker of an error location in SV-COMP. In general, the slicer can use calls to arbitrary selected functions as slicing criteria.

The tool `sbt-slicer` can be found at:

https://github.com/staticafi/sbt-slicer

[2] https://github.com/open-source-parsers/jsoncpp

5 Experimental Evaluation

This section is divided into two parts. First, we evaluate the impact of instrumentation optimizations (RC) and (RT) and slicing on the performance of Symbiotic. The second part provides a closer comparison of Symbiotic with the other two medallists in the *MemSafety* category of SV-COMP 2018, namely the tools PredatorHP [23], and UKojak [37].

In both parts, we use 390 memory safety benchmarks from SV-COMP 2018[3], namely 326 benchmarks from the *MemSafety* category and another 64 benchmarks of the subcategory *TerminCrafted*, which was not included in the official competition. The benchmark set consists of 140 unsafe and 250 safe benchmarks. The unsafe benchmarks contain exactly one error according to the official SV-COMP rules. All experiments were performed on machines with *Intel(R) Core(TM) i7-3770* CPU running at 3.40 GHz. The CPU time limit for each benchmark was set to 300 seconds and the memory limit was 4 GB. We used the utility *Benchexec* [4] for reliable measurement of consumed resources.

5.1 Contribution of Instrumentation Optimizations and Slicing

We evaluated 6 setups of the approach presented in this paper. More precisely, we consider three different configurations of instrumentation referred as *basic*, *(RC)*, and *(RC)+(RT)*, each with and without slicing. The *basic* instrumentation is the one described in Section 2. The configuration *(RC)* uses only the optimizations presented in Subsection 3.2, while the configuration *(RC)+(RT)* applies also the optimization presented in Subsection 3.3. We do not consider the configuration *(RT)* as the (RT) optimization would have hardly any effect without the (RC) optimizations.

The results are presented in Table 1 and Figure 8. The numbers of inserted calls in the table show that the extended pointer analysis itself can guarantee safety of approximately 86% of all dereferences. There is a small improvement over the conference paper [7] where the extended pointer analysis applying strong updates only for pointer assignments guaranteed about 85% of all dereferences safe. The pointer analysis itself can decide that *all* dereferences in a benchmark are safe in 103 cases. Further, nearly 42% of the dereferences that are not guaranteed to be safe can be instrumented with a simpler check instead of the expensive `check_pointer`. The optimization (RT) reduces the number of inserted memory-tracking calls to around 5%.

The numbers of instructions show that (RT) not only reduces the instrumented program size, but also substantially improves efficiency of program slicing. Altogether, all instrumentation improvements and slicing reduce the total size of programs to almost precisely 50% comparing to the basic instrumentation with slicing, and to approximately 23% comparing to the basic instrumentation without slicing.

Obviously, the most important information is the numbers of solved benchmarks. We can see that all setups detected almost all unsafe benchmarks. This

[3] `https://github.com/sosy-lab/sv-benchmarks/`, revision tag `svcomp2018` with an additional commit `514e387c` that fixes a bug in one of the benchmarks.

Table 1 For each instrumentation configuration, the table shows the total numbers of inserted calls to selected instrumentation functions, the sum of inserted calls to all checks (Σ `check_*`), and the sum of inserted calls to all tracking functions (Σ `remember_*`). Further, it shows the total numbers of instructions in instrumented benchmarks (as sent to KLEE) with and without slicing, together with their ratio in the column *relative size*. Finally, the table shows the numbers of solved benchmarks with and without slicing.

			basic	**(RC)**	**(RC)+(RT)**
inserted calls	`check_pointer`		33219	2622	2622
	`check_fail`		0	67	67
	`check_bounds`		0	425	425
	`check_heap`		0	226	226
	`check_stack`		0	638	638
	`check_globals`		0	520	520
	`check_leaks`		389	389	212
	Σ `check_*`		33608	4887	4710
	`remember_heap`		371	371	371
	`remember_stack`		11721	11721	251
	`remember_globals`		2032	2032	149
	Σ `remember_*`		14124	14124	771
number of instructions	without slicing		712428	689449	616485
	with slicing		333939	303367	166515
	relative size		47%	44%	27%
solved benchmarks	without slicing	safe	75	78	81
		unsafe	137	138	138
		total	**212**	**216**	**219**
	with slicing	safe	75	84	128
		unsafe	136	137	138
		total	**211**	**221**	**266**

confirms the generic observation that for verification tools, finding a bug is usually easier than verifying the correctness of the program. The configurations *basic* and *(RC)* solved one more unsafe benchmark without slicing than with slicing. This is because the slicer runs out of memory on this benchmark.

The situation is different for safe benchmarks. All considered setups verified between 75 and 84 safe benchmarks except *(RC)+(RT)* with slicing, which verified 128 benchmarks. This performance gap is also well illustrated by Figure 8. The lines clearly show that even though the instrumentation improvements help on their own in the end, it is the combination of (RC), (RT), *and* program slicing that helps considerably.[4]

[4] The reader may notice a difference to the conference paper in the number of solved safe benchmarks. This difference is caused by removing a heuristic we have previously added to KLEE and that turned out to be incorrect in some cases (although not on the SV-COMP benchmarks).

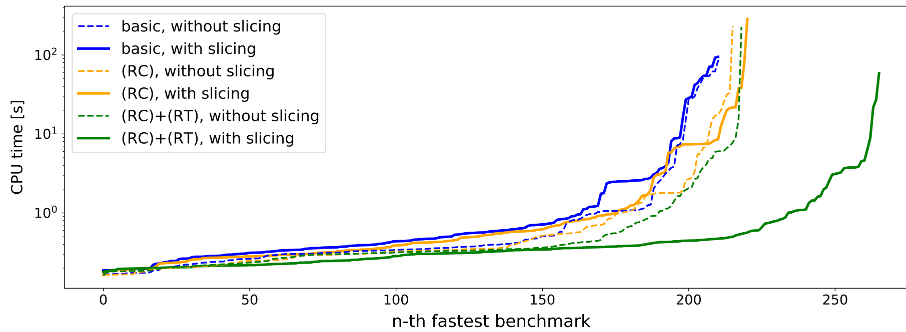

Fig. 8 Quantile plot of running times of the considered setups (excluding timeouts and errors). The plot depicts the number of benchmarks (x-axis) that the tool is able to solve in the given configuration with the given time limit (y-axis) for one benchmark.

5.2 Comparison of Symbiotic, PredatorHP, and UKojak

Now we take a closer look at the performance of the top three tools in *MemSafety* category of SV-COMP 2018, namely Symbiotic, PredatorHP [23], and UKojak [37]. We used exactly the versions of PredatorHP and UKojak that participated in the competition.

Table 2 shows the numbers of solved safe and unsafe benchmarks in each subcategory of *MemSafety* and cumulative CPU time in seconds. The row *CPU time (solved benchmarks)* gives the running time on all benchmarks that the tool solved, whereas the row *CPU time (solved by all)* provides the running time on benchmarks that were solved by all three tools. None of the tools reported any incorrect answer. Moreover, all unsafe benchmarks solved by UKojak were also solved by Symbiotic. PredatorHP solved one unsafe benchmark that Symbiotic was not able to solve (timeouted). The table shows that PredatorHP is better in solving safe instances of *Heap* and *LinkedLists* subcategories and UKojak is better in solving safe benchmarks from *Arrays* and *TerminCrafted* subcategories. Let us note that while Symbiotic and UKojak are general purpose verification tools, PredatorHP is a highly specialized tool for shape analysis of C programs that operate with pointers and linked lists. In particular, it uses an abstraction allowing to represent unbounded heap-allocated structures, which is something that at least Symbiotic cannot handle.

Further, Figure 9 provides scatter plots comparing performance of Symbiotic against the other two tools on individual benchmarks. On the left, one can immediately see that running times of UKojak are usually longer than these of Symbiotic. The fact that UKojak is written in Java and starting up the Java Virtual Machine takes time can explain a fixed delay, but not the entire speed difference. Moreover, there are 140 benchmarks solved by Symbiotic and unsolved by UKojak, compared to only 38 benchmarks where the situation is the other way around.

The plot on the right shows that PredatorHP outperforms Symbiotic on simple benchmarks solved by both tools within one second where slicing and code optimizations are redundant. Further, there are 36 benchmarks that Symbiotic was not able to solve and which were successfully solved by PredatorHP. On the

Table 2 Numbers of benchmarks in individual subcategories solved by the three considered tools. The last two rows shows the total CPU time in seconds that the tool spent on all solved benchmarks and the total CPU time that the tool spent on benchmarks that were solved by all tools, respectively.

subcategory	number of benchmarks	Symbiotic solved	Symbiotic safe/unsafe	PredatorHP solved	PredatorHP safe/unsafe	UKojak solved	UKojak safe/unsafe
Arrays	69	21	1/20	7	0/7	**39**	22/17
Heap	180	**146**	56/90	145	63/82	40	20/20
LinkedLists	51	27	3/24	**43**	19/24	1	0/1
Other	26	**26**	23/3	17	15/2	23	23/0
TerminCrafted	64	46	45/1	46	45/1	**61**	60/1
total	390	**266**	128/138	258	142/116	164	125/39
CPU time (solved benchmarks)		**266**		1349		4181	
CPU time (solved by all)		**36**		410		2284	

other hand, Symbiotic decided 44 benchmarks that were not decided by PredatorHP. For many of these benchmarks, PredatorHP gave up very quickly as its static analysis finished but was not able to decide. Moreover, many benchmarks were solved by Symbiotic within a second whereas PredatorHP computed much longer. To sum up, it seems that the benefits of Symbiotic and PredatorHP are complementary to a large extent.

Finally, Figure 10 depicts the quantile plot of running times of the three tools. Again, the plot shows that UKojak is much slower than the other two tools.

6 Advantages and Disadvantages of Our Approach

Our approach brings a tuned combination of static analysis, instrumentation, and program slicing that can greatly reduce the analysed program. One advantage of our approach is that the instrumented and reduced program can be processed by various verification methods or tools. The experiments presented in the previous section show that the achieved reduction has a signficant positive impact on the performance of symbolic execution and we expect that it may have similar effect on performance of other verification methods or tools.

Another advantage of our approach is that it transforms the problem of checking memory safety into the reachability problem. Therefore, any tool that can decide reachability can be used to verify the instrumented and reduced program.

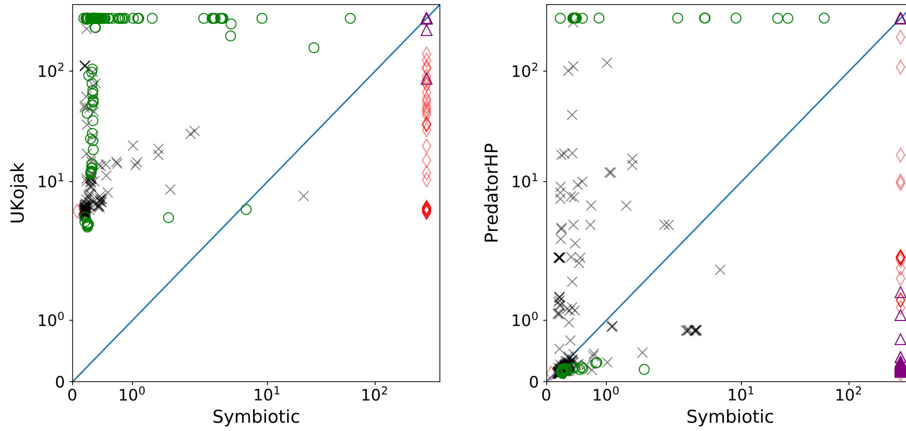

Fig. 9 Scatter plots comparing Symbiotic with UKojak (left) and with PredatorHP (right) by their running times (in seconds) on individual benchmarks. The symbols × represent benchmarks solved by both tools, ○ are benchmarks solved by Symbiotic but not by the other tool, ◇ are benchmarks solved by the other tool but not by Symbiotic, and △ are benchmarks that were solved by neither of the tools.

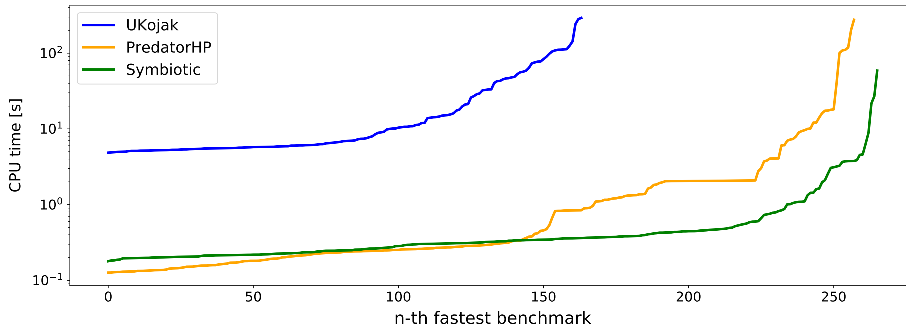

Fig. 10 Quantile plot of running times of the considered tools (excluding timeouts and errors). The plot depicts the number of benchmarks (x-axis) that the particular tool is able to solve with the given time limit (y-axis) for one benchmark.

The program can be even compiled and run, provided that the original program was compilable.

On the other hand, instrumentation inserts code manipulating complex data structures, which may be challenging for some verification tools. Here we gain by using the symbolic executor Klee that handles all the code that searches the lists with records about memory using concrete values (the search through the elements in the list is concrete, but the test whether a record is the one that we search may involve symbolic expressions). Further, since we only track what memory blocks are allocated, but we do not track the structure of the memory, we cannot reveal errors that stem from, for example, unaligned access to memory within a structure.

As our approach uses symbolic execution, we are not able to verify programs that contain unbounded heap structures or possibly unbounded loops. Symbolic

execution simply does not terminate for such programs. This inconvenience can be solved by using a different verification backend that supports analysis of programs with such traits.

As mentioned in Section 4.3, our implementation of slicing may remove some infinite loops and thus lead to false positives. This problem is just technical and does not affect the principles of this work.

7 Related Work

There are plenty of papers regarding compile-time instrumentation for detecting memory errors, but very little that optimize this process for the context of software verification: most of these papers focus on runtime monitoring and dynamic testing. Nevertheless, the basic principles and ideas of instrumentation are shared no matter whether the instrumented code is executed or passed to a verification tool. Therefore, we give an overview of tools that perform *compile-time* instrumentation although they do not verify but rather monitor the code. At the end of this section, we present an overview of tools for verification of memory safety that use some kind of instrumentation.

7.1 Runtime Monitoring Tools

Our instrumentation process is similar to the one of Kelly and Jones [25] or derived approaches like [13, 41]. The difference is that we do not need to instrument also every pointer arithmetic (as explained in Section 2) and we use simple singly-linked lists instead of splay trees to store records about allocated memory.

A different approach than remembering state of the memory in records is taken by Tag-Protector [42]. This tool keeps records and a mapping of memory blocks to these records only during the instrumentation process (the resulting program does not maintain any lookup table or list of records) and inserts ghost variables into the program to keep information needed for checking correctness of memory accesses (e.g., size and base addresses of objects). These variables are copied along with associated pointers. We believe a similar technique could be used to speed up our approach.

AddressSanitizer [43] is a very popular plugin for compile-time instrumentation available in modern compilers. It uses shadow memory to keep track of the program's state and it is highly optimized for direct execution.

To the extent of our knowledge, none of the above-mentioned approaches use static analysis to reduce the number or the runtime cost of inserted instructions.

CCured [36] is a source-to-source translator for C programming language that transforms programs to be memory safe and uses static analysis to reduce the complexity of inserted runtime checks. Static analysis is used to divide pointers into three classes: *safe*, *sequential*, and *wild* pointers, each of them deserving gradually more expensive tracking and checking mechanism. CCured does not use a lookup table but extends the pointer representation to keep also the metadata (the so-called "fat" pointers). The static analysis used by CCured is less precise as it uses unification-based approach as opposed to our analysis which is inclusion-based. Therefore, our analysis can prune the inserted checks more aggressively.

NesCheck [33] uses very similar static analysis as CCured to reduce the number of inserted checks, but does not transform the pointer representation while instrumenting. Instead, it keeps metadata about pointers separately in a dense, array-based binary search tree.

SAFECode [14] is an instrumentation system that uses static analyses to reduce the number of runtime checks. In fact, the authors also suggest to use this reduction in the context of verification. SAFECode does not try to eliminate tracking of memory blocks as our tool does. On the other hand, it employs automatic pool allocation [29] which makes lookups of metadata fast.

SoftBounds [34] is a compile-time transformation designed to check for spatial memory errors (e.g., out-of-bound access). However, it has been combined with CETS [35], which is a compile-time instrumentation system for checking for temporal memory errors (e.g., double-free). This combination is able to catch all the common memory safety errors [44].

As far as we known, the idea of using pointer analysis to reduce the fragment of memory that needs to be tracked was discussed only by Yong and Horwitz [47]. Even though the high-level concept of this work seems similar to our approach, the authors focus on runtime protection against exploitation of unchecked user inputs.

7.2 Memory Safety Verification Tools

In this subsection, we move from runtime memory safety checkers to verification tools. Instrumentation is common in this context as well, but using static analysis to reduce the number of inserted checks has not caught as much attention as we believe it deserves.

Modern verification tools support checking memory safety usually through some kind of instrumentation, but the instrumented functions are typically interpreted directly by the tool (they are not implemented in the program). CPAchecker [3] and UltimateAutomizer [20] insert checks for correctness of memory operations directly into their internal representation. SMACK [6] instruments code on LLVM level by inserting a check (that is interpreted inside the tool) before every memory-manipulating instruction.

Map2Check [39] is a memory bug hunting tool that instruments programs to track the state of allocated memory (the instrumentation is similar to Jones and Kelly's approach) and then uses verification to find possible errors in memory operations. It had used bounded model checking as the verification backend, but it has switched to a combination of fuzzing and symbolic execution (using KLEE) recently [32, 31].

None of the hitherto mentioned tools use static analysis to reduce inserted checks neither program slicing to reduce the analyzed code. More precisely, CPAchecker implements some kind of program slicing, but as far as we know, it is not applied in the standard settings.

One of few publications that explore possibilities of combination of static analysis and memory safety verification is due to Beyer et. al. [2], where authors apply CCured to instrument programs and then verify them using BLAST. The main goal was to eliminate as much inserted checks as possible using model checking.

SeaHorn [18] instruments code on LLVM level. It uses ghost variables and shadow memory to keep information needed for checking the validity of memory accesses. Checks are inserted directly into code as assertions. An unification-based field-sensitive pointer analysis is used to rule out trivial out-of-bound checks.

CBMC [27] is a bounded model checker that injects memory safety checks into its internal code representation. Checking its source code reveals that it uses a kind of lightweight field-insensitive taint analysis to reduce the number of inserted checks.

SANTE (Static ANalysis and TEsting) [10,9] is a bug hunting tool that combines static analysis, slicing, and concolic execution. The tool can find division-by-zero errors, out-of-bound array accesses and some cases of invalid pointer dereference [9]. SANTE does not use instrumentation, however, its workflow is very similar to the one of Symbiotic. It runs value analysis to reveal possible errors and then slices the program with respect to these possible errors. A specialized slice is generated for either each possible error or for some subset of the possible errors (to compare, Symbiotic always generates one slice for all possible errors). Each of the slices is then passed to concolic execution engine that checks whether the error is real. In the case of SANTE, the combination of the three techniques also proved to be more efficient and more successful than using each technique independently [10]. Unfortunately, it seems that SANTE is not maintained anymore and we were not able to get a working instance of this tool for comparison.

8 Conclusion

We presented an approach for checking memory safety properties of programs which is based on a combination of instrumentation with extended pointer analysis, static program slicing, and symbolic execution. We explained why and how we need to extend a pointer analysis and how the extended analysis can be used to reduce the number of inserted checks and to use cheaper checks in some situations. We introduced an instrumentation optimization that allows us to dramatically reduce also the number of tracked memory blocks. These instrumentation enhancements combined with static program slicing resulted in much faster analysis of error location reachability performed by symbolic execution. We implemented this technique in the tool Symbiotic that is able to compete with state-of-the-art memory safety verification tools.

Acknowledgments

The research is supported by The Czech Science Foundation grant GA18-02177S. The authors would like to thank three anonymous reviewers of STTT for their useful suggestions.

References

1. Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, 1994.

2. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Checking memory safety with blast. In *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3442 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2005.
3. Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.
4. Dirk Beyer, Stefan Löwe, and Philipp Wendler. Benchmarking and resource measurement. In *Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings*, volume 9232 of *Lecture Notes in Computer Science*, pages 160–178. Springer, 2015.
5. Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.
6. Montgomery Carter, Shaobo He, Jonathan Whitaker, Zvonimir Rakamarić, and Michael Emmi. SMACK software verification toolchain. In *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE) Companion*, pages 589–592. ACM, 2016.
7. Marek Chalupa, Jan Strejček, and Martina Vitovská. Joint forces for memory safety checking. In María-del-Mar Gallardo and Pedro Merino, editors, *Model Checking Software - 25th International Symposium, SPIN 2018, Malaga, Spain, June 20-22, 2018, Proceedings*, volume 10869 of *Lecture Notes in Computer Science*, pages 115–132. Springer, 2018.
8. Marek Chalupa, Martina Vitovská, and Jan Strejček. Symbiotic 5: Boosted instrumentation (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, volume 10806 of *Lecture Notes in Computer Science*, pages 442–446. Springer, 2018.
9. Omar Chebaro, Pascal Cuoq, Nikolai Kosmatov, Bruno Marre, Anne Pacalet, Nicky Williams, and Boris Yakobowski. Behind the scenes in sante: a combination of static and dynamic analyses. *Automated Software Engineering*, 21(1):107–143, 2014.
10. Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. The SANTE tool: Value analysis, program slicing and test generation for C program debugging. In *Tests and Proofs - 5th International Conference, TAP 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings*, volume 6706 of *Lecture Notes in Computer Science*, pages 78–83. Springer, 2011.
11. Clang: a C language family frontend for LLVM. `http://clang.llvm.org`, 2018.
12. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 25–35. ACM, 1989.
13. Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 162–171. ACM, 2006.
14. Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: enforcing alias analysis for weakly typed languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 144–157. ACM, 2006.
15. Nurit Dor, Michael Rodeh, and Mooly Sagiv. Detecting memory errors via static pointer analysis (preliminary experience). In *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '98, pages 27–34. ACM, 1998.
16. Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Byte-precise verification of low-level list manipulation. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, pages 215–237. Springer, 2013.

17. Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. In *International Symposium on Programming, 6th Colloquium, Toulouse, April 17-19, 1984, Proceedings*, volume 167 of *Lecture Notes in Computer Science*, pages 125–132. Springer, 1984.
18. Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015.
19. Samuel Z. Guyer and Calvin Lin. Error checking with client-driven pointer analysis. *Science of Computer Programming*, 58(1):83 – 114, 2005.
20. Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 36–52. Springer, 2013.
21. Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, pages 54–61. ACM, 2001.
22. Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21:848–894, 1999.
23. Lukáš Holík, Michal Kotoun, Petr Peringer, Veronika Šoková, Marek Trtík, and Tomáš Vojnar. Predator shape analysis tool suite. In *Proceedings of HVC 2016*, volume 10028 of *Lecture Notes in Computer Science*, pages 202–209. Springer, 2016.
24. Susan Horwitz, Thomas W. Reps, and David W. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.
25. Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG*, pages 13–26, 1997.
26. James C. King. Symbolic execution and program testing. *Communications of ACM*, 19(7):385–394, 1976.
27. Daniel Kroening and Michael Tautschnig. CBMC - C bounded model checker - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 389–391. Springer, 2014.
28. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, CGO '04, pages 75–88. IEEE Computer Society, 2004.
29. Chris Lattner and Vikram Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. *SIGPLAN Not.*, 40(6):129–142, 2005.
30. The LLVM compiler infrastructure. `http://llvm.org`, 2017.
31. Map2check tool. `https://map2check.github.io/`, 2018.
32. Rafael Menezes, Herbert Rocha, Lucas C. Cordeiro, and Raimundo S. Barreto. Map2check using LLVM and KLEE - (competition contribution). In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, volume 10806 of *Lecture Notes in Computer Science*, pages 437–441. Springer, 2018.
33. Daniele Midi, Mathias Payer, and Elisa Bertino. Memory safety for embedded devices with nesCheck. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 127–139. ACM, 2017.
34. Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 245–258. ACM, 2009.
35. Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. *SIGPLAN Not.*, 45(8):31–40, 2010.

36. George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. *SIGPLAN Not.*, 37(1):128–139, 2002.
37. Alexander Nutz, Daniel Dietsch, Mostafa Mahmoud Mohamed, and Andreas Podelski. ULTIMATE KOJAK with memory safety checks - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 458–460, 2015.
38. Noam Rinetzky and Shmuel Sagiv. Interprocedural shape analysis for recursive programs. In *Compiler Construction, 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2027 of *Lecture Notes in Computer Science*, pages 133–149. Springer, 2001.
39. Herbert O. Rocha, Raimundo S. Barreto, and Lucas C. Cordeiro. Hunting memory bugs in C programs with map2check - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 934–937. Springer, 2016.
40. Raphael Ernani Rodrigues, Victor Hugo Sperle Campos, and Fernando Magno Quintão Pereira. A fast and low-overhead technique to secure programs against integer overflows. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*, pages 33:1–33:11. IEEE Computer Society, 2013.
41. Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA*, pages 159–169. The Internet Society, 2004.
42. Ahmed Saeed, Ali Ahmadinia, and Mike Just. Tag-protector: An effective and dynamic detection of out-of-bound memory accesses. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, CS2 '16, pages 31–36. ACM, 2016.
43. Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28. USENIX Association, 2012.
44. Softbound + cets: Complete and compatible full memory safety for c. `https://www.cs.rutgers.edu/~santosh.nagarakatte/softbound/`, 2018.
45. Martina Vitovská, Marek Chalupa, and Jan Strejček. SBT-instrumentation: A tool for configurable instrumentation of LLVM bitcode, 2018. arXiv:1810.12617.
46. Yimin Xia, Jun Luo, and Minxuan Zhang. Detecting memory access errors with flow-sensitive conditional range analysis. In *Embedded Software and Systems: Second International Conference, ICESS 2005, Xi'an, China, December 16-18, 2005. Proceedings*, volume 3820 of *Lecture Notes in Computer Science*, pages 320–331. Springer, 2005.
47. Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pages 307–316. ACM, 2003.