Joint Forces for Memory Safety Checking

Marek Chalupa(✉), Jan Strejček, and Martina Vitovská

Masaryk University, Brno, Czech Republic
{xchalup4,strejcek,xvitovs1}@fi.muni.cz

Abstract. The paper describes a successful approach to checking computer programs for standard memory handling errors like invalid pointer dereference or memory leaking. The approach is based on four well-known techniques, namely pointer analysis, instrumentation, static program slicing, and symbolic execution. We present a particular very efficient combination of these techniques, which has been implemented in the tool Symbiotic and won by a large margin the *MemSafety* category of SV-COMP 2018. We explain the approach and provide a detailed analysis of effects of particular components.

1 Introduction

A popular application of formal methods in software development is to check whether a given program contains some common defects like assertion violations, deadlocks, race conditions, or memory handling errors. In this paper, we focus on the last mentioned group consisting of the following types of errors:

– invalid dereference (e.g. `null` pointer dereference, use-after-free)
– invalid deallocation (e.g. double free)
– memory leak

We present the approach to memory safety checking of sequential C programs implemented in Symbiotic [7], the winner of the *MemSafety* category of SV-COMP 2018. The official competition graph in Fig. 1 shows that Symbiotic (represented by the rightmost line) won by a considerable margin. One can also see that the tool is impressively fast: it would win even with its own time limit lowered to 1 s for each benchmark (the competition time limit was 900 s).

In general, our approach to memory safety checking combines static data-flow analysis with compile-time instrumentation. Static data-flow analyses for memory safety checking [11,14,35] proved to be fast and efficient. However, they typically work with under- or over-approximation and thus tend to produce false alarms or miss some errors. Instrumentation, usually 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.

The research is supported by The Czech Science Foundation, grant GA18-02177S.

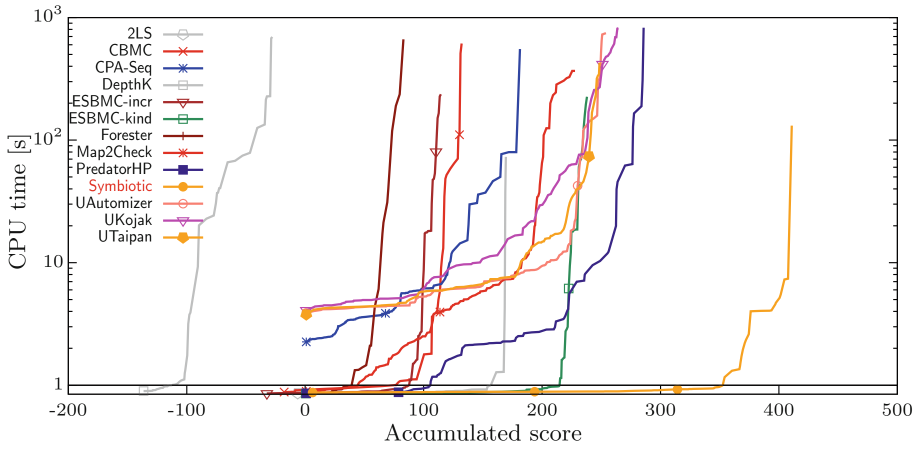

Fig. 1. The quantile plot taken from https://sv-comp.sosy-lab.org/2018/results/ representing the results of SV-COMP 2018 in the category *MemSafety*. For each tool, the plot shows what accumulated score would the tool achieve if the time limit for checking a single benchmark is set to a given time. The scoring schema assigns 1 point for every detected error provided that the tool generates an error witness which is confirmed by an independent witness checker, 1 point for the verification of a safe program (and 1 additional point if a correctness witness is generated and confirmed), and a high penalty (-16 or -32 points) for an incorrect answer. Further, the overall score is weighted by the size of subcategories. Precise description can be found at: https://sv-comp.sosy-lab.org/2018/rules.php

We combine both approaches along with static program slicing to get a reduced instrumented program that contains a reachable error location if and only if the original program contained a memory safety error. Reachability analysis is then performed to reveal possible errors in manipulation with the memory. This is the most expensive step of our approach.

The basic schema of our approach is depicted in Fig. 2. First, the program is instrumented. The instrumentation process has been augmented such that it reduces the amount of inserted code with the help of a data-flow analysis, namely an extended form of pointer analysis. We reduce the inserted code by the following three improvements:

(I1) 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.

(I2) If the pointer analysis cannot guarantee safety of a pointer dereference, but it says that the pointer refers into a memory block of a fixed known size, we insert a simpler check that the dereference is within the bounds of the block.

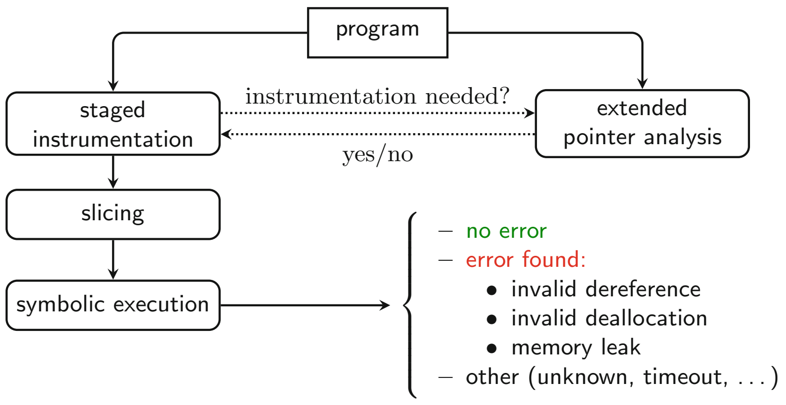

Fig. 2. The schema of our approach.

(I3) We track only information about memory blocks that can be potentially used by some of the inserted checks.

Note that interconnecting the instrumentation with a pointer analysis is not completely straightforward. Since typical pointer analyses do not care whether a memory block was freed or its lifetime has ended, a 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 needed to extend pointer analysis such that it takes into account information about freeing heap-allocated memory and the lifetime of local variables. Due to (I3), 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 tracking information about the relevant blocks.

After instrumentation, the 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.

Our instrumentation extends the program with real working code, not just with calls to undefined functions that are to be interpreted inside a verifier tool. The advantage of instrumenting the whole machinery for checking memory safety into the analyzed program (instead of inserting calls to place-holder functions interpreted by a verifier or instead of monitoring the memory inside the tool) is that the program is extended in 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 search trees for tracking the state of memory blocks.

The approach is implemented in the tool Symbiotic, which builds upon the LLVM framework [22,24]. Hence the 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 few of the LLVM instructions: `alloca` instruction allocates memory 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 other operand. Finally, `call` instruction is used to call a given function. When there is any other instruction used in the paper, its semantics is described in relevant places in the text.

This paper focuses mainly on the instrumentation part, as we use a standard static program slicing based on dependency graphs [12] and a standard symbolic execution [20]. The rest of the paper is organized as follows. Section 2 describes the basic version of code instrumentation for checking memory safety that does not use any auxiliary analysis. Section 3 introduces the extended pointer analysis and explains the instrumentation improvements (I1)–(I3). Section 4 is devoted to the implementation of our approach in Symbiotic. Section 5 presents experimental results comparing Symbiotic with state-of-the-art tools for memory safety checking and illustrating the contribution of instrumentation improvements and program slicing to the overall performance. Related work is discussed in Sect. 6.

2 Basic Instrumentation

To check memory safety, our basic instrumentation inserts a code that tracks all allocated memory blocks (including global and stack variables) and checks all memory accesses at run-time. Similarly as Jones and Kelly [19], for every allocated block of memory we maintain a record with its address and size. The records are stored in three lists:

- *StackList* for blocks allocated on the stack
- *HeapList* for blocks allocated on the heap
- *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 more precise error descriptions. For example, the information in this list enables us to distinguish double free from generic invalid deallocation, or use-after-free from vague invalid dereference error.

To maintain the three lists, after each allocation we call one of the functions `remember_stack(`*addr*, *size*`)` or `remember_heap(`*addr*, *size*`)` or `remember_global(`*addr*, *size*`)`. Before every deallocation, we call function `handle_free(`*addr*`)` that checks that *addr* points to the beginning of a memory block allocated on the heap and removes the corresponding record from

```
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);
   check_pointer(%p, 8)
6. %tmp = load %p                          %tmp = load %p
   check_pointer(%tmp, 4)                  check_pointer(%tmp, 4)
7. store i32 1 to %tmp                     store i32 1 to %tmp
```

Fig. 3. 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 improvement (I1) described in Sect. 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.

HeapList. Since local variables on the stack are destroyed when a function finishes, we call function `destroy_stack()` to remove relevant records from *StackList* right before returning from a function. 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. Finally, we insert `check_leaks()` at the end of `main` function to check that *HeapList* is empty.

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 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 [9,19,31]. We do not instrument pointer arithmetic 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 Fig. 3 (left). Allocations on lines 1 and 3 are instrumented with calls to `remember_stack` and `remember_heap`, respectively. The address of the memory allocated by the call to `malloc` is stored to `%p` on line 4. This memory is then freed and `handle_free` is called in reaction to this event. The call of `check_pointer` before line 7 reveals use-after-free error as the value of `%p` loaded on line 6 is the address of the memory allocated on line 3 and freed on line 5.

3 Instrumentation Improvements

All suggested instrumentation improvements 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 for each pointer variable its *points-to set* containing all *memory locations* 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 allocating them. For example, the object allocated on line 3 in Fig. 3 is represented by memory location `3:malloc(20)` referring to the function call that allocates the memory and its line number. Note that one memory location can represent more objects, for example when the allocation is within a program loop. Besides the 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 any referenced memory location.

The precision of pointer analysis can be tuned in several directions. We focus on *flow-sensitivity* and *field-sensitivity*. A pointer analysis is called *flow-sensitive* [16] 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 a pointer that holds at any control location in the program. For instance, in Fig. 3 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 programs are in the SSA form [8] where every program variable is assigned at a single program location only. 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. 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 Fig. 3. 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 [16,29]), but these are too expensive for our use case. Instead, we extended a simple flow- and field-sensitive Andersen's style [1] pointer analysis so that it can track whether a pointer variable can possibly point to an invalidated memory (i.e. a memory that was freed or its lifetime ended). In such a case, it includes `invalidated` in its points-to set. The extension is straightforward. Whenever the pointer analysis passes the end of a function, we traverse the points-to information and to all pointers that may point to a local object we add the `invalidated` element. Similarly, when `free` is called, we add `invalidated` element to the points-to set of pointers that may point to the freed object.

More formally, the extended pointer analysis assigns to every pointer variable p the corresponding points-to set

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

where Mem is the set of memory locations and $\mathit{Offset} = \mathbb{N}_0 \cup \{?\}$ is the set of non-negative integers extended with a special element '?' denoting an unknown offset. In the following, we assume that the information computed by the pointer analysis is *sound*, i.e. every address that can be assigned to a pointer variable p during runtime has a corresponding element in $\mathit{ptset}(p)$ (where `unknown` pointer covers any address).

(I1) Reduce the Number of Checks

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

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

where $\mathit{size}(A)$ denotes the minimum 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 $n \geq 1$).

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 Fig. 3 the dereferences of the variable `%p` on lines 2, 4, and 6 are safe and thus need not be instrumented with a check. However, we insert a check before line 7 because the analysis says that `%tmp` may point to an invalidated memory. Figure 3 (right) provides the example code instrumented using the improvement (I1).

```
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)
4. store 1 to %tmp
```

Fig. 4. Instrumentation of a code using the simpler, constant-time check. Recall that we assume that the width of an integer (`i32`) is 4 bytes.

(I2) Simplify Checks When Possible

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 a linear complexity with respect to the number of records in the lists. Here we present an improvement that can sometimes replace this check with a simpler, constant-time check.

Let us assume that there is an instruction accessing n bytes at the memory pointed by 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. Instead of a possibly expensive call `check_pointer(`p_1, n`)`, we can insert a simpler check if we know the size of the memory block referred by p_0 and where precisely p_0 points into the object (i.e. its offset). Formally, we insert the simpler check before a potentially unsafe dereference of p_1 if

– $\mathit{ptset}(p_0) \cap \{\texttt{unknown}, \texttt{null}, \texttt{invalidated}\} = \emptyset$ and
– there exist $\mathit{size}_0 > 0$ and $\mathit{offset}_0 \neq ?$ such that, for each $(A, \mathit{offset}) \in \mathit{ptset}(p_0)$, it holds that $\mathit{size}(A) = \mathit{size}_0$ and $\mathit{offset} = \mathit{offset}_0$.

Indeed, in this case we can compute the actual offset of p_1 as $\mathit{offset}_1 = \mathit{offset}_0 + (p_1 - p_0)$ and we know the size of the object that p_1 points into. The dereference is safe iff all the accessed bytes are within the bounds of the memory object, i.e. $0 \leq \mathit{offset}_1$ and $\mathit{offset}_1 + n \leq \mathit{size}_0$. This constant-time check is implemented by the function `check_bounds(`$p_1, n, p_0, \mathit{offset}_0, \mathit{size}_0$`)`.

Figure 4 provides an example where the simpler check is applied before the last instruction. In this example, an array of ten integers is allocated on line 1. 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. However, it can determine that `%array` points to the beginning (i.e. at offset 0) of the block of the size 40. Hence, the call to `check_bounds` is inserted instead of the usual `check_pointer`.

(I3) Extension with Staged Instrumentation

Although the previous two instrumentation improvements eliminate or simplify checks of dereference safety, the approach still tracks all memory allocations. However, it is sufficient to track only memory blocks that are relevant for some check. For example, the code in Fig. 3 (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 pointer checked by `check_pointer(%tmp, 4)` can never point to the location `1:alloca i32*`. Hence, the call to `remember_stack` inserted after line 1 can be safely omitted. Note that we always track all allocations on heap as they are relevant for the memory leaks checking.

In order to insert only relevant calls to `remember_stack` and `remember_global` functions, we perform the instrumentation in two stages.

1. In the first stage, checks are inserted as described before. Additionally, for every inserted `check_pointer` call, we remember its first argument, i.e. the pointer variable. In the first stage, we also insert all calls to `remember_heap`, `handle_free`, and `destroy_stack`.
2. The second stage inserts calls to `remember_stack` and `remember_global`. For every memory location A corresponding to a global variable or some allocation on the stack, we check whether any pointer variable remembered in the first stage can point into the memory location A. Formally, we check that there exists some remembered pointer p such that $(A, \mathit{offset}) \in \mathit{ptset}(p)$ for some offset, or $\texttt{unknown} \in \mathit{ptset}(p)$. We insert the call to `remember_stack` or `remember_global` only if the answer is positive. 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.

Note that in the first stage we do not remember arguments of `check_bounds` introduced by (I2) as this function does not search the lists of records.

In Figs. 3 (right) and 4, the presented staged instrumentation would not insert any call to `remember_stack`.

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 program slicing before the reachability analysis.

4 Implementation

The described approach was implemented in Symbiotic [7]. The tool consists of three main parts, namely *instrumentation module*, *slicing module* and the external state-of-the-art open-source symbolic executor Klee [5]. Moreover, 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 Sect. 3.

Instead of implementing a single-purpose instrumentation for memory safety checking, we developed a configurable instrumentation module [34]. The instrumentation process is controlled by a configuration file provided by the user. A configuration can specify an arbitrary number of instrumentation stages, each defined by a set of instrumentation rules. Every rule describes which instructions should be matched and how to instrument them. At the moment, the instrumentation can insert only `call` instructions as it is sufficient for most use-cases. An instrumentation rule can trigger an additional action like setting a flag or remembering values or variables used by the matched instructions. Further, a 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 already remembered (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 due to (I1).

Besides a configuration, the user has to also provide definitions of functions whose calls are inserted into the program. For checking memory safety, these functions are written in C and translated to LLVM. After a successful instrumentation, these functions are linked to the instrumented code.

We implemented a static backward slicing algorithm based on dependence graphs [12] as we have not found any suitable program slicer for LLVM bitcode. The algorithm has been extended to a simple form of inter-procedural slicing, where dependence graphs for procedures are connected by inter-procedural edges and the slice is obtained by one backward search instead of using the traditional two-pass algorithm introduced in [18].

Symbiotic applies code optimizations provided by the LLVM framework after instrumentation and again after slicing. Finally, Klee is executed on the sliced and optimized code to check for reachability of the inserted error locations.

The tool Symbiotic and its components are licensed under the MIT and Apache-2.0 open-source licenses and can be found at:

https://github.com/staticafi/symbiotic

Klee is licensed under the University of Illinois license.

5 Experimental Evaluation

The section is divided into two parts. First, we compare several setups of the described approach in order to show which ingredients are essential for good performance. The second part provides a closer comparison of Symbiotic with the other two winning tools in the *MemSafety* category of SV-COMP 2018.

Table 1. For each instrumentation configuration, the table shows the total numbers of inserted calls of `check_pointer`, `check_bounds`, and `rememeber*` functions. 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.

	Inserted calls			Number of instruct.			Solved benchmarks			
	`check_pointer`	`check_bounds`	`remember*`	w/o slicing	with slicing	relative size	w/o slicing		with slicing	
							safe	unsafe	safe	unsafe
Basic	32333	0	10511	575k	343k	60%	116	132	118	131
(I1)	4930	0	10511	538k	303k	56%	119	132	125	132
(I1) + (I2)	4750	180	10511	538k	301k	56%	119	132	126	132
(I1) + (I3)	4930	0	830	478k	174k	36%	130	132	180	132
(I1) + (I2) + (I3)	4750	180	792	478k	171k	36%	132	132	181	132

5.1 Contribution of Instrumentation Improvements and Slicing

We evaluated 10 setups of the approach presented in this paper. More precisely, we consider five different configurations of instrumentation referred as *basic*, *(I1)*, *(I1)+(I2)*, *(I1)+(I3)*, and *(I1)+(I2)+(I3)*, each with and without slicing. The *basic* instrumentation is the one described in Sect. 2 and the other four configurations employ the corresponding improvements presented in Sect. 3. We do not consider other configurations as they are clearly inferior.

For the evaluation, we use 390 memory safety benchmarks from SV-COMP 2018[1], namely 326 benchmark from the *MemSafety* category and another 64 benchmarks of the subcategory *TerminCrafted*, which was not included in the official competition this year. 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 s and the memory limit was 6 GB. We used the utility *Benchexec* [4] for reliable measurement of consumed resources.

The results are presented in Table 1 and Fig. 5. The numbers of inserted calls in the table show that the extended pointer analysis itself can guarantee safety of approximately 85% of all dereferences. In other words, (I1) reduces the number of inserted checks to 15%. Further, (I2) can replace a relatively small part of these checks by simpler ones. The improvement (I3) reduces the number of inserted memory-tracking calls to around 8% in both configurations *(I1)+(I3)* and *(I1)+(I2)+(I3)*.

The numbers of instructions show that (I3) 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 30% 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 132 unsafe benchmarks except the

[1] https://github.com/sosy-lab/sv-benchmarks/, revision tag `svcomp2018`.

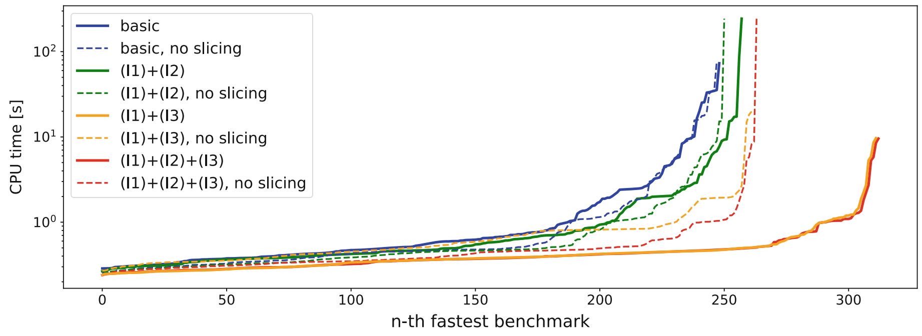

Fig. 5. Quantile plot of running times of the considered setups (excluding timeouts and errors). The plot depicts the number of benchmarks (x-axis) that the tool is able to solve in the given configuration with the given time limit (y-axis) for one benchmark. We omitted the lines for *(I1)* with and without slicing as they almost perfectly overlap with the corresponding lines for *(I1)+(I2)*.

basic configuration with slicing, where the slicing procedure did not finish for one benchmark within the time limit. As the considered benchmark set contains only 140 unsafe benchmarks, this confirms the generic observation that for verification tools, finding a bug is usually easy. The situation is different for safe benchmarks. All considered setups verified between 116 and 132 safe benchmarks except *(I1)+(I3)* with slicing and *(I1)+(I2)+(I3)* with slicing, which verified 180 and 181 benchmarks, respectively. This performance gap is also well illustrated by Fig. 5. The lines clearly show that even though the instrumentation improvements help on their own, it is the combination of (I1), (I3) *and* program slicing that helps considerably. The effect of (I2) is rather negligible.

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 [17], and UKojak [28]. What we present and interpret are the official data of this category available on the competition website https://sv-comp.sosy-lab.org/2018/. Note that SV-COMP 2018 used 900 s timeout and memory limit of 15 GB per benchmark.

Table 2 shows the numbers of solved safe and unsafe benchmarks in each subcategory and total time needed to solve these benchmarks. None of the tools reported any incorrect answer. Symbiotic was able to solve the most benchmarks (almost 80%) in very short time compared to the other two tools. Moreover, all unsafe benchmarks solved by PredatorHP and UKojak were also solved by Symbiotic. PredatorHP is better in solving safe instances of *Heap* and *LinkedLists* 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 operating with pointers and linked lists.

Table 2. Numbers of bechmarks solved by the three considered tools in each subcategory of *MemSafety*. The last row shows total CPU time spent on all solved benchmarks.

		Symbiotic		PredatorHP		UKojak	
subcategory	number of benchmarks	solved	$\frac{\text{safe}}{\text{unsafe}}$	solved	$\frac{\text{safe}}{\text{unsafe}}$	solved	$\frac{\text{safe}}{\text{unsafe}}$
Arrays	69	**62**	$\frac{44}{18}$	7	$\frac{0}{7}$	44	$\frac{27}{17}$
Heap	180	145	$\frac{55}{90}$	**148**	$\frac{66}{82}$	51	$\frac{26}{25}$
LinkedLists	51	27	$\frac{3}{24}$	**43**	$\frac{19}{24}$	4	$\frac{0}{4}$
Other	26	**26**	$\frac{23}{3}$	18	$\frac{16}{2}$	23	$\frac{23}{0}$
total	326	**260**	$\frac{125}{135}$	216	$\frac{101}{115}$	122	$\frac{76}{46}$
CPU time [s]		**310**		2100		11000	

In particular, it uses an abstraction allowing to represent unbounded heap-allocated structures, which is something at least Symbiotic cannot handle.

Scatter plots in Fig. 6 provide another comparison of the tools. On the left, one can immediately see that running times of UKojak are much longer than these of Symbiotic for nearly all benchmarks. 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 141 benchmarks solved by Symbiotic and unsolved by UKojak, compared to only 3 benchmarks where the situation is the other way around. In many of these cases, UKojak gave up or crashed even before time limit.

The plot on the right shows that PredatorHP outperforms Symbiotic on simple benchmarks solved by both tools within one second. Further, there are 34 benchmarks where Symbiotic timed out but which were successfully solved by PredatorHP. On the other hand, Symbiotic decided 78 benchmarks that were not decided by PredatorHP. For most of these benchmarks, PredatorHP gave up very quickly as its static analysis is unable 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.

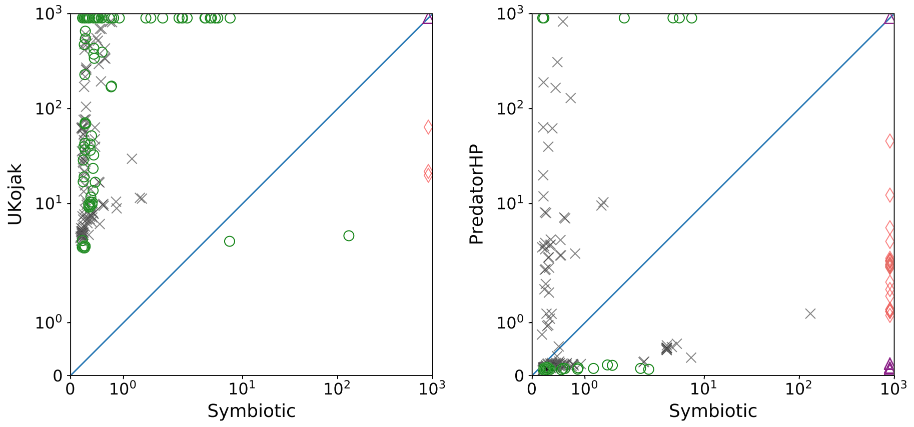

Fig. 6. Scatter plots comparing Symbiotic with UKojak (left) and with PredatorHP (right) by their running times (in seconds) on individual benchmarks. The symbols × represent benchmarks solved by both tools, ○ are benchmarks solved by Symbiotic but not by the other tool, ◇ are benchmarks solved by the other tool but not by Symbiotic, and △ are benchmarks that were solved by neither of the tools.

6 Related Work

There is plenty of papers on runtime instrumentation for detecting memory errors, but very little that optimize this process for the context of software verification. Nevertheless, the basic principles and ideas 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. Further, an overview of tools for verification of memory safety is also provided.

6.1 Runtime Monitoring Tools

Our instrumentation process is similar to the one of Kelly and Jones [19] or derived approaches like [9,31]. The difference is that we do not need to instrument also every pointer arithmetic (as explained in Sect. 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 [32]. 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 insert 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 [33] 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 [27] 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 expansive 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 opposed to our analysis which is inclusion-based. Therefore, our analysis can prune the inserted checks more aggressively.

NesCheck [26] 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 pointer separately in a dense, array-based binary search tree.

SAFECode [10] is an instrumentation system that uses static analyses to reduce the number of runtime checks. In fact, they also suggest to use this reduction in the context of verification. SAFECode does not try to eliminate the tracking of memory blocks as our tool does. However, it employs automatic pool allocation [23] to make lookups of metadata faster.

As far as we known, the idea of using pointer analysis to reduce the fragment of memory that needs to be tracked appeared only in [36]. Even though the high-level concept of this work seems similar to our approach, they focus on runtime protection against exploitation of unchecked user inputs.

6.2 Memory Safety Verification Tools

In the rest of this section, 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 also support checking memory safety usually through some kind of instrumentation, but the instrumented functions are interpreted directly by the tool (they are not implemented in the program). CPAchecker [3] and UltimateAutomizer [15] insert checks for correctness of memory operations directly into their internal representation. SMACK [6] and SeaHorn [13] instrument code on LLVM level. SeaHorn uses ghost variables for checking out-of-bound memory accesses via assertions inserted into code, and shadow memory to track other types of errors. SMACK inserts a check before every memory access. Map2Check [30] is a memory bug hunting tool that instruments programs and then uses verification to find possible errors in memory operations. It used bounded model checking as the verification backend, but it has switched to LLVM and KLEE recently [25]. All these tools use no static analysis to reduce inserted checks.

One of few publications that explore possibilities of combination of static analysis and memory safety verification is [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.

Finally, CBMC [21] injects 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.

7 Conclusion

We have presented a technique for checking memory safety properties of programs which is based on a combination of instrumentation with extended pointer analysis, program slicing, and symbolic execution. We describe how the extended pointer analysis can be used to reduce the number of inserted checks and showed that in some cases these checks can be further simplified. We introduced an instrumentation improvement that allows us to dramatically reduce also the number of tracked memory blocks. These instrumentation enhancements combined with program slicing result in much faster analysis of error location reachability that is performed by symbolic execution. We implemented this technique in the tool SYMBIOTIC that has consequently won the *MemSafety* category of Software Verification Competition 2018 and thus proved to be able to compete with state-of-the-art memory safety verification tools.

References

1. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. Ph.D thesis, DIKU, University of Copenhagen (1994)
2. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: Checking memory safety with blast. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 2–18. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31984-9_2
3. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
4. Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement. In: Fischer, B., Geldenhuys, J. (eds.) SPIN 2015. LNCS, vol. 9232, pp. 160–178. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23404-5_12
5. Cadar, C., Dunbar, D., Engler, D.R.: 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, 8–10 December 2008, San Diego, California, USA, Proceedings, pp. 209–224. USENIX Association (2008)
6. Carter, M., He, S., Whitaker, J., Rakamarić, Z., Emmi, M.: SMACK software verification toolchain. In: Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE) Companion, pp. 589–592. ACM (2016)

7. Chalupa, M., Vitovská, M., Strejček, J.: SYMBIOTIC 5: boosted instrumentation. In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10806, pp. 442–446. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_29
8. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: 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, 11–13 January 1989, pp. 25–35. ACM (1989)
9. Dhurjati, D., Adve, V.: Backwards-compatible array bounds checking for C with very low overhead. In: Proceedings of the 28th International Conference on Software Engineering, ICSE 2006, pp. 162–171. ACM (2006)
10. Dhurjati, D., Kowshik, S., Adve, V.: SAFECode: enforcing alias analysis for weakly typed languages. In: PLDI 2006: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 144–157. ACM (2006)
11. Dor, N., Rodeh, M., Sagiv, M.: 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 1998, pp. 27–34. ACM (1998)
12. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. In: Paul, M., Robinet, B. (eds.) Programming 1984. LNCS, vol. 167, pp. 125–132. Springer, Heidelberg (1984). https://doi.org/10.1007/3-540-12925-1_33
13. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_20
14. Guyer, S.Z., Lin, C.: Error checking with client-driven pointer analysis. Sci. Comput. Program. **58**(1), 83–114 (2005)
15. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 36–52. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_2
16. Hind, M.: 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 2001, Snowbird, Utah, USA, 18–19 June 2001, pp. 54–61. ACM (2001)
17. Holík, L., Kotoun, M., Peringer, P., Šoková, V., Trtík, M., Vojnar, T.: Predator shape analysis tool suite. In: Bloem, R., Arbel, E. (eds.) HVC 2016. LNCS, vol. 10028, pp. 202–209. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49052-6_13
18. Horwitz, S., Reps, T.W., Binkley, D.W.: Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. **12**(1), 26–60 (1990)
19. Jones, R.W.M., Kelly, P.H.J.: Backwards-compatible bounds checking for arrays and pointers in C programs. In: AADEBUG, pp. 13–26 (1997)
20. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
21. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_26

22. Lattner, C., Adve, V.: 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 2004, pp. 75–88. IEEE Computer Society (2004)
23. Lattner, C., Adve, V.: Automatic pool allocation: Improving performance by controlling data structure layout in the heap. SIGPLAN Not. **40**(6), 129–142 (2005)
24. The LLVM compiler infrastructure (2017). http://llvm.org
25. Map2check tool (2018). https://map2check.github.io/
26. Midi, D., Payer, M., Bertino, E.: Memory safety for embedded devices with nesCheck. In: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, ASIA CCS 2017, pp. 127–139. ACM (2017)
27. Necula, G.C., McPeak, S., Weimer, W.: CCured: type-safe retrofitting of legacy code. SIGPLAN Not. **37**(1), 128–139 (2002)
28. Nutz, A., Dietsch, D., Mohamed, M.M., Podelski, A.: ULTIMATE KOJAK with memory safety checks. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 458–460. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_44
29. Rinetzky, N., Sagiv, M.: Interprocedural shape analysis for recursive programs. In: Wilhelm, R. (ed.) CC 2001. LNCS, vol. 2027, pp. 133–149. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45306-7_10
30. Rocha, H.O., Barreto, R.S., Cordeiro, L.C.: Hunting memory bugs in C programs with Map2Check. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 934–937. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_64
31. Ruwase, O., Lam, M.S.: A practical dynamic buffer overflow detector. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA, pp. 159–169. The Internet Society (2004)
32. Saeed, A., Ahmadinia, A., Just, M.: 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 2016, pp. 31–36. ACM (2016)
33. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Addresssanitizer: a fast address sanity checker. In: Proceedings of the 2012 USENIX Conference on Annual Technical Conference, USENIX ATC 2012, pp. 28–28. USENIX Association (2012)
34. Vitovská, M.: Instrumentation of LLVM IR. Master's thesis, Masaryk University, Faculty of Informatics, Brno (2018)
35. Xia, Y., Luo, J., Zhang, M.: Detecting memory access errors with flow-sensitive conditional range analysis. In: Yang, L.T., Zhou, X., Zhao, W., Wu, Z., Zhu, Y., Lin, M. (eds.) ICESS 2005. LNCS, vol. 3820, pp. 320–331. Springer, Heidelberg (2005). https://doi.org/10.1007/11599555_32
36. Yong, S.H., Horwitz, S.: 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, pp. 307–316. ACM (2003)