CheckPointer

A C Memory Access Validator

Michael Mehlich
Semantic Designs, Inc.
Austin, TX
mmehlich@semanticdesigns.com

Abstract—**CheckPointer is a memory access validator for checking spatial and temporal pointer usage errors in multi-threaded applications by tracking meta data and validating pointer dereferences at run-time. The tool uses source-to-source transformations implemented with DMS to instrument the source code of the application to be validated with meta data checks. Libraries available only in binary form are handled by using function wrappers that check meta data immediately before calling a library function and update meta data as necessary immediately after the library function returns.**

Keywords-CheckPointer; memory safety; memory debugger; out-of-bounds error; pointer error; memory access error; instrumentation; source-to-source transformations; DMS

I. Introduction

Many security vulnerabilities and software bugs in C have their origin in pointer usage errors such as accessing memory outside the bounds of the current object or accessing memory that is not currently allocated by the application. Where these errors cause immediate deterministic application failures, their location is usually easy to pinpoint. However, often the application will continue executing for some time after the original error has been made, before delayed symptoms of the errors can be observed, e.g. in form of the application following some unexpected code paths, producing unexpected results, or causing an access violations in a different code fragment. In general, the delayed symptoms have no obvious connection to the original error, making it extremely hard to determine their cause, i.e. to pinpoint the original error. Thus, a tool reporting the original error immediately when performed is invaluable for the engineer debugging and testing such applications.

The CheckPointer memory access validator provides a means to get such early reporting of pointer usage errors. The tool achieves this by instrumenting the source code to track additional information (a.k.a. "meta data") about objects and pointers in the application and validate any pointer usage for well-definedness against this information. The pointer usage errors detected by the tool include the following:

- *Spatial errors:*
  Access of memory outside the bounds of a known (sub-)object.
  - *Invalid pointer error:*
    Dereference of a pointer that is null, uninitialized, or has been computed incorrectly from an arbitrary source.
  - *Out-of-bounds error:*
    Use of a subscript that is outside the bounds of an array or string (sub-)object, or
    Dereference of a pointer resulting in access of memory outside the bounds of a (sub-)object. This includes the case where dereferencing a pointer to a member field results in accessing memory outside the bounds of the member field though still within the object.
  - *Invalid free error:*
    Deallocation of non-heap object, e.g. a global or local object or a string literal, or
    Deallocation of sub-object of a heap-allocated object.
  - *Wrong kind of pointer error:*
    Read or write though a (type-casted) function pointer, or call through a (type-casted) data pointer.
- *Temporal errors:*
  Access of memory that has already been deallocated.
  - *Deallocated heap error:*
    Dereference of a pointer to heap (sub-)object that has already been deallocated.
  - *Escaped pointer error:*
    Dereference of a pointer to a non-static local (sub-)object in a block after execution of the block has finished.
  - *Double free error:*
    Deallocation of a heap-allocated object that has already been deallocated.

  These temporal errors are detected even if the deallocated memory is being reused for a different stack- or heap-allocated object but accessed through a stale pointer.

To increase responsiveness, to take advantage of multi-core processors, or to simplify coding of cooperative tasks, applications increasingly exploit multiple threads of execution. In order to deal with such applications, the source code instrumented by the CheckPointer memory access validator performs thread-safe updates of meta data[*], tracks thread local data, and checks for related *escaped pointer* errors, i.e. for dereferences of pointers *to thread-local storage* of threads that already terminated.

II. Implementation

CheckPointer is implemented on top of the DMS Software Reengineering Toolkit [2], a generic compiler infrastructure for the development of source code analysis and transformation tools, and its front-end for the C programming language.

The infrastructure and front-end provide means to preprocess and parse C source code in various dialects, perform name and type resolution to construct a proper symbol table for the source code, modify the resolved source code by applying source-to-source transforms that may query the symbol table and other semantic information, and unparse the modified source code for further processing by external tools, e.g. a compiler.

CheckPointer augments this infrastructure and front-end by providing source-to-source transforms that insert meta data tracking and pointer dereference validating statements and by orchestrating the instrumentation of multiple translation units, allowing for incremental instrumentation of modified translation units without the needs to reinstrument the whole application.

A. *Organization of Meta-data*

In order to validate all memory accesses in the application code, the instrumented source code produced by CheckPointer keeps track of the following meta data:

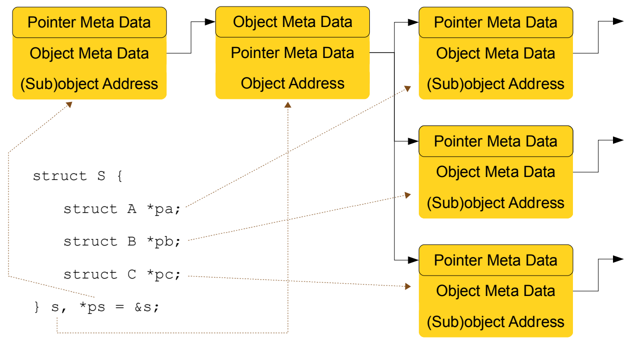
- Pointer meta data for each pointer, consisting of
    - A reference to the object meta data for the heap-allocated object or global or local variable or function pointed to by the pointer and
    - The address range of the (sub)object of the object that the pointer may currently access. This may be smaller than the address range of the whole object; e.g. if you take the address of a struct member, the instrumented source code will only allow access to that member when using the resulting pointer.
- Object meta data for each heap-allocated object, for each variable and function the address of has been taken, and for each variable of struct or array type containing member fields or array elements of pointer type, consisting of
    - The kind and location of the object, i.e. whether it is a function, a global, thread-local or local variable, heap-allocated memory, or a string literal constant,
    - The address range of the object that may be safely accessed, and
    - For each pointer stored in the heap-allocated object or variable, a reference to the pointer meta data for that pointer.

Whenever a pointer is assigned to another pointer, the associated pointer meta data needs to be copied, too. This includes the case where a struct is assigned to another struct, in case of which the pointer meta data for all member fields of pointer type must be copied accordingly.

Whenever the lifetime of an object ends, its associated object meta data, if exists, must be destroyed and all pointers pointing to or into the object must be marked as stale in order to check for temporal errors. However, CheckPointer does not keep track of all the pointer meta data referencing a particular object meta data. Thus, in order to achieve the desired effect, the object meta data is only marked as "destroyed" but its memory is not deallocated. Object meta data memory is recycled for other object meta data by using a unique "generation key" stored in both the object meta data and each pointer meta data referencing the object meta data. Using this approach, "destruction" of object meta data consists of updating the generation key, and validating a pointer includes performing a non-staleness check consisting of checking whether the generation key stored in its associated pointer meta data is identical to the generation key stored in the referenced object meta data.

Figure 1 shows the meta data for an example, a pointer `ps` to a struct value `s` containing three pointers `pa`, `pb`, and `pc`. The pointer `ps` has associated pointer meta data referencing the object meta data associated to the struct value `s`. The object meta data for `s` references the pointer meta data associated to three member pointers `pa`, `pb`, and `pc`.

If a new pointer value is assigned to `ps`, then the pointer meta data associated to the new pointer value is copied into the pointer meta data associated to `ps`. Similarly, if a new pointer value is assigned to a pointer member of `s`, say `pb`, then the pointer meta data associated to the new pointer value is copied into the pointer meta data associated to the member `pb` in the object meta data associated to `s`. If a new



```
struct S {
    struct A *pa;
    struct B *pb;
    struct C *pc;
} s, *ps = &s;
```

Figure 1. Organization of meta data (example).

[*] CheckPointer assumes that the application does not contain any data races, i.e. read/write or write/write "conflicts" between parallel threads, though on many hardware platforms read/write accesses to e.g. pointer variables are actually atomic. If such data races exist, the meta data may not be read/written properly despite the original accesses being atomic, which may eventually cause memory access errors being reported or missed. If a "data race" is intentionally used by a particular application, the corresponding read/write accesses must be protected by proper locking in the original code in order to properly execute the instrumented code. Preprocessing conditions can be used to ensure that locking only occurs in the instrumented code. Note that the C standard does not require read/write accesses to pointer variables being atomic.

```
rule ReplaceAR2a4a1(id1:IDENTIFIER,id3:IDENTIFIER,id4:IDENTIFIER):expression->expression
  = "\id1 = &\id3->\id4"
 -> "(
      // compute pointer meta data for \id1
      pointer_meta_data_restrict(&\GetPointerMetaData\(\id1\), &\GetPointerMetaData\(\id3\),
                                 &\id3->\id4, sizeof \id3->\id4),
      // perform original assignment
      \id1 = &\id3->\id4
     )"
 if IsPointerObjectOrParameter(id1) /\ IsPointerObjectOrParameter(id3).
```

Figure 2. Instrumentation rule for `id1 = &id3->id4`.

value is assigned to `s`, the pointer meta data associated to all the pointer members, i.e. to `pa`, `pb`, and `pc` are copied. If `s` ceases to exist, e.g. because it was a local variable and the function defining `s` terminated, the object meta data associated to `s` is marked as "destroyed" causing all pointers to `s` considered as being stale.

To pass meta data between a calling and a called function, the instrumented source code manages a shadow stack of call meta data. For each function call, the shadow stack contains the following information:

- The number of actual arguments passed to the called function (assigned by the calling function),
- For each pointer formal parameter of the called function the pointer meta data of the actual argument (assigned by the calling function),
- For each struct or union formal parameter of the called function, the pointer meta data for the address of the actual argument (assigned by the calling function and immediately used by the called function to copy the referenced object meta data to the object meta data of the formal parameter),
- For a pointer result of the called function, the pointer meta data of the called function's result (assigned by the called function),
- For a struct or union result of the called function, the pointer meta data for the address of the target location for the function result (assigned by the calling function with the referenced object meta data being updated by the called function upon execution of a return statement).
- Position information for the function call (used to create a call trace when an error is encountered), and
- A reference to the call meta data for the parent call.

A pointer to the top entry of the shadow stack is stored in a global or thread-local variable. Passing meta data between the calling and the called function therefore does not require changing function signatures.

The call meta data is allocated on the actual C call stack, which avoids heap allocation of the call meta data but may require increasing the stack size for the application.

B. *Instrumentation of Source Code*

CheckPointer realizes run-time tracking of meta data and validation of pointer dereferences by instrumenting the original source code using source-to-source transformations implemented using the DMS Software Reengineering Toolkit and its C front-end supporting various C dialects, including GCC and Visual C.

Each translation unit of an application is instrumented separately using the following basic steps:

- Preprocess and parse the source code into a syntax tree,
- Perform name and type resolution over the syntax tree to create a symbol table,
- Normalize the syntax tree into a functionally equivalent simplified form of C,
- Instrument the normalized syntax tree using source-to-source transformations, and
- Unparse the instrumented syntax tree into instrumented source code.

For each processed translation unit the instrumenter also augments a cache file containing information about the files used in the application, the meta data for global variables needed or made available by the translation unit, the sizes of global array variables, and the sizes of flexible array members of global struct variables. After all translation units of the application have been instrumented, this information is used to generate an auxiliary source file containing declarations and missing initializers of meta data and an auxiliary header file containing the collected sizes of the arrays and flexible array members. The auxiliary header file is needed to perform proper out-of-bounds checks on these arrays and array members in translation units that do not explicitly specify their sizes.

1) Sample Instrumentation Rules: During instrumentation all pointer dereferences must be instrumented in order to validate the well-definedness of the respective dereferences and propagate the corresponding pointer meta data when necessary. Using the DMS Software Reengineering Toolkit, such instrumentation can be achieved by modifying the (name and type resolved) syntax tree searching for patterns describing pointer dereferences and replacing these with appropriate validating and propagating code fragments. Though performed on the syntax tree level, such replacement rules are actually specified using the native syntax of the underlying programming language C embedded in a meta language for declaring rules and combining them to rule sets.

Figure 2 shows a transformation rule for dealing with assignments of the form `id1 = &id3->id4` where `id1` is a variable of pointer type.

```
rule ReplaceAR3a5a1(id1:IDENTIFIER,id3:IDENTIFIER,id4:IDENTIFIER):expression->expression
  = "*\id1 = \id3->\id4"
 -> "(
      // check for *\id1 being writable
      pointer_meta_data_check_write(\id1, sizeof *\id1,
                                &\GetPointerMetaData\(\id1\), \GetPosition\(*\id1\)),
      // check for \id3->\id4 being readable
      pointer_meta_data_check_read(&\id3->\id4, sizeof *&\id3->\id4,
                               &\GetPointerMetaData\(\id3\), \GetPosition\(\id3->\id4\)),
      // propagate pointer meta data for \id3->\id4 to pointer meta data for *\id1
      *pointer_meta_data_lookup(\GetPointerMetaData\(\id1\).object_meta_data, \id1)
        = *pointer_meta_data_lookup(\GetPointerMetaData\(\id3\).object_meta_data, &\id3->\id4),
      // perform original assignment
      *\id1 = \id3->\id4
     )"
 if    IsPointerObjectOrParameter(id1) /\ IsPointerExpression("*\id1")
    /\ IsPointerObjectOrParameter(id3) /\ IsPointerMemberField(id4).
```

Figure 3. Instrumentation rule for `*id1 = id3->id4`.

The right hand side of the rule consists of two steps, first assigning the pointer meta data for the pointer to the member field to the pointer meta data associated to the target, which is computed by restricting the pointer meta data for the pointer `id3` to the address range of the field `id4`, and then executing the original assignment. This rule does not introduce any validation for pointer dereferences as the original code only computes the address of a field without dereferencing any pointer.

Figure 3 shows a transformation rule for dealing with assignments of the form `*id1 = id3->id4` where all involved variables and member fields are of pointer type.

The comments on the right hand side of the rule indicate the steps performed by the replacement of the original expression, consisting of first validating whether the target of the assignment can be written and whether the source of the assignment can be read, then copying the pointer meta data associated to the source pointer to the pointer meta data associated to the target pointer, which involves looking up the pointer meta data via the corresponding pointed-to objects, and finally executing the original assignment.

For local variables the address of which is taken, the instrumenter needs to create and "destroy" object meta data at appropriate points during program execution.

Figure 4 shows a transformation rule[*] for dealing with the `break` statement. This rule's application condition (implemented by procedural code walking over the syntax tree and consulting the symbol table) checks whether terminating the nearest enclosing loop or switch statement ends the lifetime of a local object declared within the loop or switch statement for which object meta data may have been created during execution. If this is the case, the rule inserts the finalizers necessary to "destroy" the object meta data associated to such local variables, ensuring that further accesses to these local variables through pointers will fail. Similar rules are provided to cover other forms of jump statements.

```
private rule FinalizersForBreak
               (ss:statements,s:statement)
             :statement->statement
   = "{ \ss break; }"
  -> "{ \ss
        \CreateFinalizersForBreak\(break;\);
        break;
      }"
  if DoesBreakRequireFinalizers("break;").
```

Figure 6. Instrumentation rule for `break` statement.

CheckPointer uses a large number of rules[*] similar to the above ones to declare pointer meta data and object meta data where necessary, validate pointers that are being dereferenced, propagate pointer meta data for assignments, mark object meta data for local variables as "destroyed" when their respective lifetimes end, and add function prologues and epilogues to create and destroy call stack meta data. The orchestrated application of these rules ensures that the resulting instrumented code is well-formed and performs the desired memory access validation checks when compiled and executed.

2) Wrappers for Binary Libraries: The instrumentation added by CheckPointer assumes that the whole application is instrumented properly to manage the necessary meta data. One way to achieve this is by instrumenting the complete source code of the application. However, in practice, this is often not possible, as the application uses functionality provided by the C standard library, by the operating system, and by third party libraries, which often only come in binary form with a description of its available interfaces. Thus, instrumented source code has to interact with non-instrumented library code.

In general, the instrumented source code can be directly linked with the non-instrumented library code as the function signatures are not changed by the instrumenter. Thus, pointers passed into functions provided by non-instrumented libraries, as well as pointers received from functions (by means of the function's return value or its side-effects) in these libraries do not pose any problems for the instrumented source

* Note that when the rule is being processed all occurrences of the break statement reference the same syntax tree node; which is the break statement in the syntax tree for the source code being instrumented the left hand side of the rule successfully matched with.

* The number of rules is positively correlated with the number of different forms of statements, in particular assignments involving pointers, remaining after normalizing the syntax tree into a functionally equivalent simplified form of C. The more forms are remaining, the more rules are needed instrument all pointer usages. However, in the absence of any other optimizations, more remaining forms allow better instrumentation code being produced.

code as long as they are handled as opaque data. However, such pointers are not validated when dereferenced from within the non-instrumented libraries. The instrumented source code also cannot dereference pointers received by non-instrumented libraries, as proper meta data for these pointers is not available.

These limitations can be overcome by producing wrappers for the non-instrumented functions. In general, such wrappers have three major phases to safely simulate the original function:

- *Validate that calling the original function is safe:*
  The function arguments and any other program state are checked to validate that calling the original function will not cause any memory access errors. This usually involves reading and checking the meta data associated to the function arguments and/or other global objects.
- *Call the original function:*
  The original function is called to ensure that the core functionally of the call is preserved by the wrapper function.
- *Update meta data to reflect the effects of the original function:*
  The pointer and object meta data tracked by CheckPointer to perform memory access validation must be updated to reflect the effects of the original function on such data. Dependent on the function this may include creating, reading, updating, and/or deleting such meta data.

Further complications arise when an instrumented function is to be passed as a call-back to a non-instrumented function. In such cases, a wrapper needs to be passed instead that updates meta data if necessary, creates appropriate call meta data before calling the actual instrumented function. and destroys the call meta data afterwards. If necessary, this may be followed by some additional validation of meta data to ensure that continuing execution in the caller is safe.

CheckPointer provides an interface to write such function wrappers by providing means to validate pointers against meta data and update meta data.

This interface has been used to implement wrappers for the standard C library functions, including the memory copy functions `memcpy` and `memmov`, which require copying all pointer meta data within an address range to corresponding pointer meta data in another address range from the object meta data of an object to the object meta data of the same or another object.

Similarly, the interface has also been used to implement a wrapper for the begin thread call in multi-threaded applications. This wrapper performs some initial setup and instead of starting a thread with the desired (instrumented) function starts the thread with a wrapper function that first establishes an appropriate context for meta data management, then calls the desired (instrumented) function, and finally upon its termination marks the object meta data for thread-local variables as being "destroyed". The wrapper for the end thread call is somewhat more involved as it needs to mark as "destroyed" not only the object meta data for thread-local variables but also for any local variables along the call stack starting from the root function of the thread. This can be achieved by the instrumented source code tracking the object meta data for local objects and thread-local variables in appropriate lists associated to the call stack meta data, which then can be accessed by the begin thread and end thread calls when needed.

The wrapping approach to thread-local variables and thread execution management relies on these capabilities being provided in form of library functions. However, such capabilities are not standardized and may be provided by other means for a particular compiler or platform. E.g. gcc provides a `__thread` keyword while Visual C provides a `__declspec(thread)` phrase as a storage class specifier to declare "global" or "static local" variables as being thread-local. In general, CheckPointer may need to be modified to

```
#include <windows.h>
#include <process.h>
#include <stdio.h>

__declspec(thread) int tls_variable;
int *global_pointer;

unsigned int __stdcall set_global_pointer(void *args) {
  global_pointer = &tls_variable; // set global pointer to thread-local variable
  *global_pointer = 1; // well-defined dereference of pointer to thread-local variable
  return 0;
}

int main() {
  //printf("Global pointer to thread local variable in current thread\n");
  set_global_pointer(0);
  *global_pointer = 2; // well-defined dereference of pointer to thread-local variable
  //printf("Global pointer to thread local variable in terminated thread\n");
  HANDLE thread = (HANDLE)_beginthreadex(NULL, 0, set_global_pointer, NULL, 0, NULL);
  WaitForSingleObject(thread, INFINITE); // wait for thread to terminate
  CloseHandle(thread);
  *global_pointer = 3; // illegal dereference of escaped pointer to thread-local variable
  return 0;
}
```

Figure 4. Erroneous multi-threaded C program.

```
PARLANSE Compiler V19.16.41
*** Error: Dereference of pointer is out of bounds.
    in function: ReadNextInputBlock, line: 697, file P0File.c
    called in function: SetUpINCLUDEFile, line: 593, file: P0File.c
    called in function: SemIncludeFile, line: 27437, file: P0Compiler.c
    called in function: LRParser, line: 8256, file: P0Compiler.c
    called in function: CompileSourceFile, line: 8824, file: P0Compiler.c
    called in function: main, line: 9254, file: P0Compiler.c
*** Error: Dereference of pointer is out of bounds.
    in function: NextFileByte, line: 718, file P0File.c
    called in function: NextSourceByte, line: 867, file: P0File.c
    called in function: GetNextTerminal, line: 6556, file: P0Compiler.c
    called in function: EnsureHoldingCurrentToken, line: 7833, file: P0Compiler.c
    called in function: LRParser, line: 8089, file: P0Compiler.c
    called in function: CompileSourceFile, line: 8824, file: P0Compiler.c
    called in function: main, line: 9254, file: P0Compiler.c
```

Figure 5. CheckPointer error messages for an instrumented compiler.

understand such compiler or platform specific constructs. Its current implementation already handles the above mentioned constructs when processing source code in the respective dialects, and utilizes these storage class specifiers to declare variables for tracking thread-local meta data.

C. Examples

CheckPointer has been successfully applied to various test cases and a commercial compiler for a proprietary parallel programming language developed in C that has been in continuous use for over a decade. CheckPointer itself has been implemented in this parallel programming language and compiled with that compiler.

1) Escaped Pointer to Thread-local Variable: The C program in figure 5 contains one of the test cases used for validating CheckPointer. In this test case, a pointer to a thread-local variable is stored in a global variable which later is dereferenced after the thread the variable was associated to terminated.

Compiled normally with Visual C++ 2010 Express in debug mode and executed in the debugger[*], this test case finishes execution of the main function apparently successfully, but afterward causes a breakpoint exception in an NT kernel function called from

```
msvcr100d.dll!_free_base(void *pBlock)
  Line 50 + 0x13 bytes
```

The only suggestion about the error given by the debugger is that the error “may be due to a corruption of the heap”, without providing any further help in pinpointing the location of the error.

If instead the test case is instrumented using CheckPointer, the instrumented source code compiled with Visual C++, and the resulting binary executed, the following output is produced:

```
*** Error: Dereference of dangling pointer.
    in function: main, line: 22, file Example.c
```

The error report clearly indicates the location of the erroneous pointer dereference in the source code before the assignment can do any damage to internal data structures.

2) Compiler for a Parallel Language: CheckPointer has been applied to a (single-threaded) compiler for a proprietary parallel programming language, called PARLANSE [1]. This compiler has been used continuously since over a decade to compile parallel programs. Applications programmed in PARLANSE and successfully compiled with the compiler include the DMS Reengineering Toolkit, including the compiler infrastructure and the C front-end used by CheckPointer, and in particular procedural parts of CheckPointer itself.

The compiler's source code, constisting of about 110,000 lines of C code, has been instrumented with CheckPointer, the instrumented source code compiled with GCC4, and the resulting binary used to compile CheckPointer itself, producing the output shown in figure 6.

Upon further investigation of the error locations, both cases turned out to be out-of-bounds accesses to an array embedded in a larger struct, although the memory accessed was still within the enclosing struct. (The compiler worked by accident.)

D. Related Work

There are various tools for detecting spatial or temporal pointer usage errors, or a combination thereof, at run-time. Such tools use different approaches for instrumenting the application:

- *Source code instrumentation* consists of adding run-time checks into the source code, in general independent of a particular compiler or target platform. The resulting instrumented source code is processed by a compiler after successful instrumentation.

  This approach is used by CheckPointer.

  CCured [8] also uses source code instrumentation, reducing overhead for tracking meta data and validating pointer usage by applying optimizations based on an extension of C's type system. However, CCured uses fat pointers causing incompatibility issues in real code due to memory layout changes of compound data types, while CheckPointer avoids layout changes by keeping meta data in separate data structures. CCured ignores explicit deallocation of memory; instead it uses a garbage collector to reclaim heap-allocated memory and disallows pointers to local variables being stored in heap-allocated memory and global variables.

* Note that if executed outside the debugger, the test case, though compiled in debug mode, appears to finish successfully without producing any error message at all.

MSCC [14] is another system using this approach, and is implemented using CIL [9] as a foundation, transforming a simplified form of C within CIL. Like CheckPointer, MSCC does not use fat pointers but instead uses shadow data associated with pointers and objects. However, MSCC only allows limited casting between pointers of "related" struct types; and does not check for out-of-bounds errors on sub-objects. MSCC does not support multi-threaded programs.

- *Compile-time instrumentation* adds run-time checks while the source code is being compiled, usually exploiting internal data structures and optimizations used by the compiler.

  SoftBound [6] and CETS [7] use this approach, with SoftBound providing spatial error checking and CETS providing temporal error checking. These tools are implemented using the LLVM compiler framework [4] and operate on LLVM's typed single static assignment form. SoftBound uses pointer meta data similar to CheckPointer, tracking pointers stored in memory using a hash table or a shadow space. CETS uses a generation key similar to CheckPointer to detect dangling pointers. The CETS prototype does not support multi-threaded programs.

  MemSafe [13] is another tool using compile-time instrumentation. It uses meta data similar to CheckPointer. However, while CheckPointer uses separate maps from pointers to pointer meta data for each variable or heap-allocated object containing pointers, MemSafe uses a single global map. MemSafe utilizes a whole program data-flow for pointers graph to perform various optimizations reducing execution overhead.

  Fail-Safe C [11] uses fat pointers similar to CCured but alleviates layout change issues by mapping from virtual to physical byte offsets when accessing memory where necessary. Local variables the address of which are taken are moved into the heap. Garbage collection is used to avoid dangling pointers to heap allocated memory, though memory is marked as non-accessible when "freed". Wrappers are needed to interact with library functions that are not compiled with Fail-Safe C to convert between the fat pointer representation and the normal representation. Unlike CheckPointer, Fail-Safe C does not check for out-of-bounds errors on sub-objects.

- *Binary instrumentation* defers adding run-time checks until after compiling the application into a binary, thus does not suffer from source code for libraries not being available. However, high-level information from the source code, e.g. the size and bounds of member fields in a struct, are in general not available in binary executables; thus tools based on binary instrumentation cannot detect out-of-bounds errors for such member fields.

  Purify [3] adds binary instrumentation on object files after source code compilation but before linking. It checks for pointer usage errors only for heap-allocated objects.

  Valgrind [10] uses on-demand dynamic recompilation of small code blocks in binary executables, by first translating a block of machine code into an intermediate representation, then allowing plug-ins to instrument this internal form, and finally converting it back to machine code that is then being executed. An available plug-in, called Memcheck, adds instrumentation for pointer usage checks. Like Purify, Valgrind does not detect out-of-bounds errors for global or local variables. Valgrind supports multi-threaded programs but serializes execution to ensure that only one thread is running at a time.

E. *Conclusion*

CheckPointer is a memory access validator for detecting spatial and temporal pointer usage errors in multi-threaded C applications at run-time. The tool is implemented using source-to-source transformations using the DMS Software Reengineering Toolkit and its C front-end as a basis. The instrumented source code produced by CheckPointer from the original source code keeps track of meta data about pointers and objects and checks all pointer dereferences for well-definedness, immediately reporting an error before it can do any actual damage. Functions for which no source code is available are dealt with by manually producing wrappers simulating the effects of the function on the meta data.

As CheckPointer intercepts all memory allocations and deallocations to manage meta data for heap allocated objects, the instrumented code can be configured to report all memory locations that are still being allocated at the end of application execution. Such information can be utilized to check for memory leaks.

The current implementation of CheckPointer could be improved by using static analysis to eliminate redundant meta data updates and checks in the instrumented source code, significantly reducing the execution time of the instrumented application. Such redundant checks include pointer dereference checks dominated by other identical pointer dereference checks, and pointer dereference checks for accessing multiple pointed-to fields of a single struct that could be summarized into a single check across all fields. The DMS Software Reengineering Toolkit and the C front-end provide supporting infrastructure and analyses to implement such optimizations.

Elapsed time could also be reduced by offloading the meta data tracking and pointer usage validation from the main thread to auxiliary parallel threads [5]. This is particularly useful for a single-threaded application that is being executed on a multi-core platform.

CheckPointer currently assumes that direct and indirect function calls are used with properly typed arguments. However, C compilers in general do not enforce this for direct function calls across translation units, nor for indirect calls through function pointers even within a single translation unit. In order to catch such errors, CheckPointer could be extended to keep track of static and dynamic type information and check for compatibility of such information where necessary to ensure safe execution.

For availability of CheckPointer see [12].

References

[1] I.D. Baxter, “Parallel support for source code analysis and modification,” International Workshop on Source Code Analysis and Manipulation, pp. 3–15, IEEE, 2002.

[2] I.D. Baxter, C. Pidgeon, and M. Mehlich, “DMS: Program transformation for practical scalable software evolution,” Proceedings of the International Conference on Software Engineering, pp. 625–634, IEEE Computer Society, 2004.

[3] R. Hastings and B. Joyce, “Purify: Fast detection of memory leaks and access errors in C and C++ programs,” Proceedings of the Winter USENIX Conference, pp. 125–138. USENIX, 1992.

[4] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis and transformation,” Proceedings of the Symposium on Code Generation and Optimization, pp. 75–86, IEEE Computer Society Press, 2004.

[5] S. Lee an J. Tuck, “Automatic parallelization of fine-grained metafunctions on a chip multiprocessor,” International Symposium on Code Generation and Optimization, 2011.

[6] S. Nagrakatte, J. Zhao, M.M.K. Martin, and S. Zdancewic, “SoftBound: Highly compatible and complete spatial memory safety for C,” Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 245–258, ACM 2009.

[7] S. Nagrakatte, J. Zhao, M.M.K. Martin, and S. Zdancewic, “CETS: Compiler-enforced temporal safety for C,” Proceedings of the International Symposium on Memory Management, ACM, 2010.

[8] G.C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, “CCured: Type-safe retrofitting of legacy code,” Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 128–139, ACM, 2002.

[9] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer, “CIL: Intermediate language and tools for analysis and transformation of C programs,” Proceedings of the International Conference on Compiler Construction, pp. 213–228, Springer, 2002.

[10] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 89–100, ACM, 2007.

[11] Y. Oiwa, “Implementation of the memory-safe full ANSI-C compiler,” ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 259–269, ACM, 2009.

[12] Semantic Designs, “CheckPointer: C memory safety checker,” Internet: `http://www.semanticdesigns.com/Products/MemorySafety/CMemorySafetyChecker.html`, 2011 [June 7, 2011].

[13] M.S. Simpson and R.K. Barua, “MemSafe: Ensuring spatial and temporal memory safety of C at runtime,” IEEE Working Conference on Source Code Analysis and Manipulation, pp. 199–208, IEEE, 2010.

[14] W. Xu, D.C. DuVarney, and R. Sekar, “An efficient and backwards-compatible transformation to ensure memory safety of C programs,” Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 117–126, ACM, 2004.