Computer Languages, Systems & Structures 45 (2016) 191–203

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Lightweight runtime checking of C programs with RTC[☆]

Reed Milewicz[a,*], Rajesh Vanka[b], James Tuck[c], Daniel Quinlan[d], Peter Pirkelbauer[a]

[a] *University of Alabama at Birmingham, United States*
[b] *Matlab, United States*
[c] *North Carolina State University, United States*
[d] *Lawrence Livermore National Laboratory, United States*

ARTICLE INFO

Article history:
Received 1 July 2015
Received in revised form
24 November 2015
Accepted 10 January 2016
Available online 15 January 2016

Keywords:
Runtime monitoring
Source code instrumentation
Static analysis
C
C++

ABSTRACT

The C Programming Language is known for being an efficient language that can be compiled on almost any architecture and operating system. However the absence of dynamic safety checks and a relatively weak type system allows programmer oversights that are hard to spot. In this paper, we present RTC, a runtime monitoring tool that instruments unsafe code and monitors the program execution. RTC is built on top of the ROSE compiler infrastructure. RTC finds memory bugs and arithmetic overflows and underflows, and run-time type violations. Most of the instrumentations are directly added to the source file and only require a minimal runtime system. As a result, the instrumented code remains portable. In tests against known error detection benchmarks, RTC found 98% of all memory related bugs and had zero false positives. In performance tests conducted with well known algorithms, such as binary search and MD5, we determined that our tool has an average run-time overhead rate of 9.7 × and memory overhead rate of 3.5 × .

1. Introduction

One major trend in computing is the continuing increase in the complexity of software systems. Such an increase is allowed by the expectation of increasingly powerful hardware (faster processors, larger memory and disks) and the increasing diversity of environments in which the software runs. This increase in complexity is expensive; the National Institute for Standards and Technology (NIST) estimated that inadequate infrastructure for software testing costs the US economy $22.2 billion annually [31].

Many programming languages allow the use of unsafe programming constructs in order to attain a high degree of flexibility and performance. This makes the construction of correct large-scale software difficult. Unsafe language features allow programmer oversights to introduce software flaws which can create security hazards that can compromise an entire system. Eliminating these software flaws can be addressed on many levels in the software engineering process. Rigid coding

[☆] This paper is an expanded version of the "Runtime Error Checking C programs" presented at SAC/PL 2015. This contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the United States Government retains a nonexclusive, royalty-free right to publish or reproduce this paper, or to allow others to do so, for Government purposes only.
[*] Corresponding author.
E-mail addresses: rmmilewi@cis.uab.edu (R. Milewicz), rvanka@ncsu.edu (R. Vanka), jtuck@ncsu.edu (J. Tuck), dquinlan@llnl.gov (D. Quinlan), pirkelbauer@uab.edu (P. Pirkelbauer).

http://dx.doi.org/10.1016/j.cl.2016.01.001
1477-8424/© 2016 Elsevier Ltd. All rights reserved.

standards, restricting the use of a large language to a safer subset, peer review, and the use of static or dynamic analysis tools are some means that can all reduce the exposure to software flaws.

Analysis tools can be categorized by how they analyze software. Static tools analyze software without running it. They target source code (occasionally binary) to apply formal analysis techniques such as dataflow analysis, abstract interpretation, and model checking; such techniques often use approximations to arrive at sound but imprecise conclusions about the behavior of programs. Dynamic analysis tools find bugs by observing the behavior of running programs. This is typically accomplished by code instrumentation (source or binary) or by replacing (built-in) library functions (e.g., malloc and free) with custom implementations. Dynamic analysis operates with concrete values and is not prone to combinatorial state explosion. The downside of dynamic analysis tools is that monitoring running programs impacts performance. The quality of the results depends on the tests' input data and covered program paths. Hybrid analyses combine static and dynamic techniques. Hybrid tools can improve performance by eliminating checks that a static analyzer considers safe in all possible scenarios, or they can improve test coverage by producing input values that cover all possible paths.

The history of program analysis research now spans several decades, and dozens of tools and techniques have pushed the envelope on our ability to detect and correct bugs. However, in the area of dynamic analysis, certain fundamental challenges remain, namely high overhead costs and lack of portability. Neither of these challenges have gone unnoted, but have been relatively low priority targets for researchers. Severe overhead, in many respects, has been seen as the cost of doing business, a burden that software developers have been willing to tolerate. For dynamic analysis tools that target binaries for instrumentation (the most popular choice), high overhead costs are difficult to avoid. The most efficient way to reduce overhead is to selectively reduce instrumentation where bugs are unlikely or are rendered impossible, but without access to high-level information (e.g. type information) to guide the pruning process, this can be a risky proposition. Meanwhile, the computing landscape has been historically dominated by only a handful of operating systems and varieties of architecture; meeting the portability requirements of developers meant being able to operate in two or three popular environments.

However, as we move towards a future where computing pervades every aspect of our daily lives, these challenges become more substantial and more must be done to address them. The most visible signal of the shift to ubiquitous computing has been the proliferation of smartphones and the thriving ecosystem of services that interface with them. However, the most influential transformations have come from the progressive infiltration of embedded computing systems, from critical control devices in medical care and avionics, to smart televisions and coffee machines. Many of the most popular languages for development in these burgeoning environments are also the least safe (e.g. C and C++), the improper use of which introduces vulnerabilities that threaten reliability and security. Of particular importance is the need to perform in situ analysis, where we are able to leverage multiple devices exposed to real-world inputs in order to expose vulnerabilities quickly and efficiently. Unfortunately, burdensome overhead costs and portability limitations render such analysis impractical if not impossible; many tools currently available are unable to keep pace with the growing demands. The imperative is a simple one: adapt or die.

In this paper, we present RTC (Run-Time error check for C programs), a dynamic analysis tool for C99 programs. RTC instruments source code with safety checks and produces another C source file. The resulting source file is portable and can be compiled on any platform and any compiler that can handle C99 and linked to a small runtime system. Choosing to instrument source code instead of binary code has a number of advantages. First, the tool is portable, because the systems where the code is instrumented and the system where the code runs can be different architectures. The only requirement is that there is a C99 compiler available for the target system. Second, by instrumenting source code, the tool processes the code as written by the programmer, and not some code that was generated and possibly optimized by a compiler. Finally, we can choose to validate only a single program module. For example, we may want to check only a single, commonly used library. In such cases, we may want to instrument only that library and not the whole application.

Currently, RTC supports C99 and a subset of sequential C++. RTC implements three kinds of safety checks: arithmetic overflow/underflow, memory safety checks to find memory bugs on stack and heap, and run-time type-safety violations. The metadata is kept on the side using a locks and keys approach. Arithmetic overflow/underflow and memory safety checks cover three of the most dangerous software bugs [29]. We tested RTC on several runtime checking benchmarks and on complete programs including grep, crafty, and other C programs in the SPEC2000 benchmark suite.

The paper presents the following contributions: (1) automated and portable source code instrumentation and monitoring for C99 programs; (2) lightweight runtime monitoring implementation.

The paper is outlined as follows: Section 2 presents background information and related work. Section 3 describes our implementation in detail, and Section 4 discusses how we tested our tool and the obtained results. Section 5 summarizes the paper and discusses possible future research directions.

2. Background

In this section, we present earlier work on error checking, and an overview of the ROSE source-to-source transformation system.

2.1. Related work

Run-time error checking tools have been designed for a variety of reasons, ranging from bug detection and security to software verification.

SafeC [1] introduced the notion of using source code instrumentation to detect temporal and spatial memory errors. Ccured [23], Cyclone [14], and MSCC [34] are all works derived from or inspired by SafeC. Of particular interest to us is CCured, which introduced the notion of using lightweight, disjoint metadata facilities to cut down on the massive overhead inherent in the approach taken by SafeC. This idea inspired Hardbound [6], which attempts to tackle the issue by giving hardware support for bounds checking pointers and pointer management. A software-based approach analogous to HardBound was explored in SoftBound [22]. MemSafe [30] extends this idea by using static analysis to prove memory accesses safe. ConSeq [36] identifies code locations, such as assertions or reads of key global variables. Then ConSeq extracts slices to determine instructions that contribute to violations. Finally, ConSeq uses dynamic analysis to generate valid, bug-free executions of the program, and then see whether any legal deviations from that execution could lead to the potential errors that were revealed by the static analysis.

RTED [28] is a dynamic analysis tool developed at the Lawrence Livermore National Laboratory. RTED is a first proof of concept implementation for sequential C, a large sequential subset of C++, and UPC[7], a parallel language for the partitioned global address space model. RTED finds temporal and spatial memory violations on stack and heap, signature mismatch in declaration and definition, erroneous library calls, and reads from uninitialized memory. RTED statically inserts source code to monitor the execution. For concurrent codes in UPC, RTED synchronizes shared memory accesses to force a deterministic execution. Consequently, the runtime overhead of RTED is large ($>100\times$). For the targeted errors, the error detection tool achieved roughly 95% coverage on Iowa State's runtime error detection benchmark suite [18].

Frama-C [16] is an optimized runtime memory monitoring library that implements assertion checking, memory and pointer safety checking as library. Frama-C is complemented by E-ACSL which is a first-order logic annotation language [5,13]. The specifications can be translated to Frama-C runtime checks.

Other verification tools are also available. Valgrind [25], a framework for writing dynamic binary analyses tools. Valgrind finds memory access violations, concurrency related bugs, and others. It supports concurrency, but serializes concurrent executions. IBM's Rationale Purify [12] is a memory debugger that instruments object code and tracks memory allocation and initializations. Parasoft's Insure++ [27] is a proprietary tool that instruments source and monitors execution in threaded applications. CDSChecker [26] is a stateless model checker for concurrent software written in the C11 and C++11 programming languages.

Other parallel error checking tools are geared to find concurrency bugs [2,9,20]. Some runtime error checking approaches require support built into hardware [4,8,19,21,37]. Probabilistic methods allow to deal with uncertainty resulting from less frequent program observation [15].

Lastly, we note that [3] explored run-time type-checking in C, by using binary instrumentation informed by debugging information to perform fine-grained type-safety analysis.

2.2. The Rose transformation system

ROSE, developed at the Lawrence Livermore National Laboratory (LLNL), is a source-to-source translation infrastructure for multiple languages, including C/C++, Fortran 77/95/2003, Java, and UPC. ROSE also supports several extensions to develop parallel programs, such as OpenMP and CUDA. ROSE represents source code as abstract syntax trees (AST). The ASTs

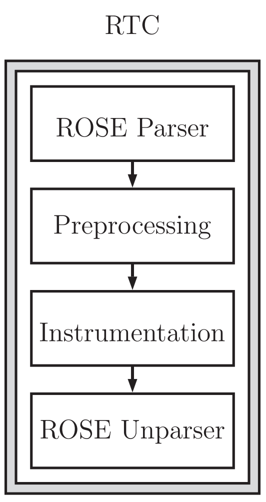

Fig. 1. The RTC approach.

are built in a uniform and consistent way for all input languages. ROSE implements many specific analyses (e.g. pointer alias analysis) and makes them available through an API. Users can write their own analysis by utilizing frameworks that ROSE provides. These include attribute evaluation traversals, call graph analysis, control flow graphs, class hierarchies, SSA representation, and dataflow analysis. ROSE has been used for building custom tools for static analysis, program optimization, arbitrary program transformation, domain-specific optimizations, performance analysis, and cyber-security. ROSE can regenerate `include` directives, which maintains the portability of original code.

3. Implementation

We shall attempt to provide an overview of the implementation of the RTC tool. First, RTC supplies the original C source code to the ROSE compiler framework, which parses the input to produce an internal intermediate representation (IR), including a detailed abstract syntax tree (AST). The IR is, in turn, provided to RTC. RTC makes a pass over the AST to identify regions of code that require monitoring (such as when memory is allocated or a pointer is returned from a function call). Our tool then instruments the code by decorating the AST. Finally, the AST is unparsed, yielding the instrumented C source code. This code can then be compiled, linked to RTC's metadata libraries, and then executed. The monitoring infrastructure put in place during the instrumentation phase allows us to probe the behavior of the program at runtime to detect bugs. A diagram illustrating this process can be seen in Fig. 1.

The following subsections provide detailed descriptions of each of the aforementioned steps.

3.1. Preprocessing

When RTC receives the AST from ROSE, the AST must be preprocessed. The purpose of the preprocessing phase is to make the AST more amenable to instrumentation while preserving the semantics of the program. By requiring that the AST be normalized first, we reduce the complexity of the instrumentation process and ensure that instrumentation is consistently and correctly applied. The preprocessing step is split up over many distinct functions, each of which queries the AST to find and perform normalizing operations on select nodes of the AST. These transformations include the following:

1. The termination conditions of for and while loops are moved inside their associated blocks. This guarantees that all for and while loops have similar structures.
2. Arrow expressions are converted to dot expressions. This allows us to treat arrow and dot expressions in the same fashion.
3. Structs defined within functions are moved out into the global scope. Otherwise, the instrumentation functions might overlook the struct definitions.

Once all has been laid bare, RTC performs a complete traversal of the AST, composing a list of every node that requires instrumentation. The resulting list is then passed to the instrumentation phase.

3.2. Instrumentation

The purpose of the instrumentation phase is to guarantee that every interaction with memory is guarded by appropriate calls to runtime monitoring functions, and that all pointers are outfitted with metadata that allows us to track when, where,

```
void process_heartbeat(unsigned char *hbMessage) {
  unsigned short hbtype;
  unsigned int payload;
  unsigned char* contents;
  hbtype = hbMessage[0];
  payload = (((unsigned int)(hbMessage[1]))<< 8)
            | (((unsigned int)(hbMessage[2])));
  hbMessage += 3;
  contents = hbMessage;
  if (hbtype == TLS1_HB_REQUEST) {
    unsigned char* response;
    response = (unsigned char *) malloc(1 + 2 + payload + padding);
    ...
    memcpy(response, pl, payload);
    ...
  }
  ...
}
```

Fig. 2. A simplified representation of the code responsible for the Heartbleed bug.

and how they are used. At the same time, the panoptical ideal can only be realized within certain constraints: we must apply the instrumentation consistently and preserve the intended behaviors of the program. The preprocessing phase guarantees the first requirement, but not the latter, which is the subject of this subsection. For the purposes of demonstrating this process, we have taken an abridged excerpt from version 1.0.1 of the OpenSSL library, containing the code responsible for the Heartbleed bug which was first disclosed in April 2014. The exploit relies on a spatial memory violation, and we shall demonstrate how RTC instruments this code to expose the violation at run-time.

The first objective of the instrumentation process is to expose all pointers so that their uses can be analyzed at runtime. For each pointer, RTC maintains a metadata record that is kept in a separate and disjoint data structure. Pointers are moved into structures that contain two fields: one for the pointer and another to hold the stack address of that pointer; the latter serves as an index to that pointer's associated metadata. For every type of pointer in the input program, RTC declares and defines a struct to hold pointers of that type, as well as functions that handle the creation of those structs. In the case of our example, RTC provides structs for pointers to types `unsigned char` and `void`. An assignment to a pointer variable is split into a declaration and a call to a specialized assignment function that creates an instance of the struct and assigns the variable. Operations that alter pointers are substituted with calls to specialized functions that perform these operations and move the pointer into new wrapper structs; pointers are kept in constant motion, donning whatever raiment is appropriate for the present circumstance.

This system of pointer-wrapping structs also forms the backbone for the runtime type-checking system, as it provides the opportunity to record the data type of the pointer. This is done by mapping the stack address of the pointer to a type history, a list of types associated with the pointer, stored in chronological order. In practice, types are represented by special typeinfo nodes, which list essential details about the type, such as whether the type is a primitive or an aggregate, the `sizeof` the type, and the base type (if one exists). RTC determines what types are used in a program statically, and injects declarations for each typeinfo node into the global scope, as well as calls to metadata library functions to instantiate the typeinfo nodes at run-time. In this way, information about each type is computed only once. This is an instance of the flyweight design pattern: type histories consist of a sequence of pointers to these typeinfo nodes, instead of having many redundant copies. Type histories are retained until a pointer is deallocated. This type history can be analyzed on demand for runtime type safety violations, as we shall see in Section 3; see Grimmer et al. [11] for a description of a similar system for runtime type-checking.

When a pointer is passed to a function, the associated stack address is pushed onto a globally accessible stack provided by the metadata libraries. Upon entering the function, the address is popped from the stack so that it and the pointer can be redressed in a new wrapper. By using an external stack, RTC avoids having to alter function signatures to make metadata information available across scopes. An important consequence of this is that it is possible to call instrumented code from uninstrumented code without having to make changes or adaptations due to syntactic mismatches – crucial for testing purposes.

Once this has been done, RTC then inserts instrumentation around code where pointers are used to check for possible violations. As seen in Fig. 4, array accesses are guarded by calls to a bounds checking function that ensures that the access falls within the lower and upper bounds of the array allocation. Checks are also placed around calls to functions including `malloc`, `free`, and `pthread_create`, and other important library functions like `memcpy`. Calls to third party library

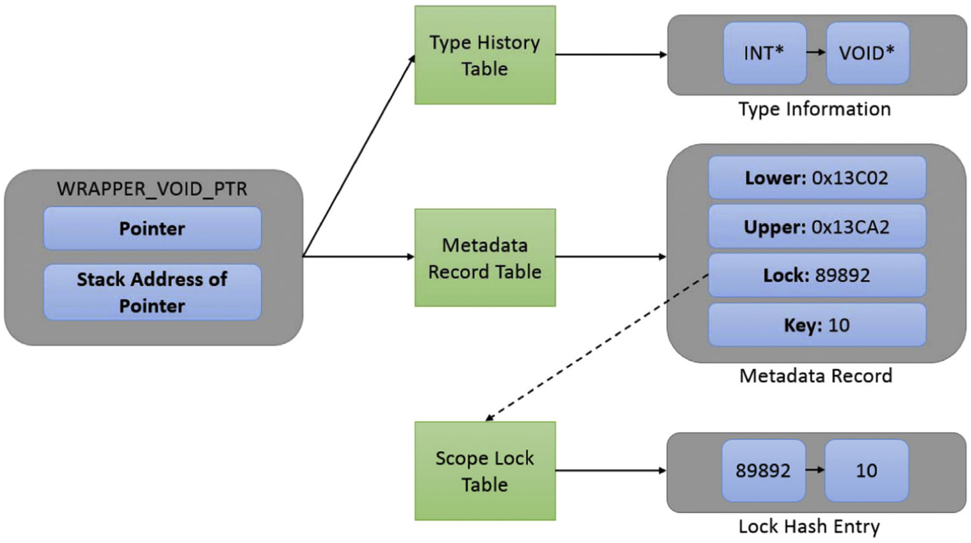

Fig. 3. A visualization of how data associated with pointers is stored at run-time. Pointers are shadowed by lightweight metadata structures that hold the stack address of the pointer, which is used to index into the metadata record table and type history table. The lock entry in the metadata record can be used to index into the scope lock table to confirm that the key is the valid one for the current scope.

```
void process_heartbeat(unsigned char *hbMessage) {
  CREATE_MD_ENTRY_IF_EXISTS(&hbMessage,GET_FROM_STACK(0));
  unsigned short hbtype;
  unsigned int payload;
  unsigned char* contents;

  ARRAY_BOUNDS_CHECK(&hbMessage,&hbMessage[0]);
  hbtype = hbMessage[0];

  ARRAY_BOUNDS_CHECK(&hbMessage,&hbMessage[1]);
  ARRAY_BOUNDS_CHECK(&hbMessage,&hbMessage[2]);
  payload = (((unsigned int)(hbMessage[1]))<< 8)
            | (((unsigned int)(hbMessage[2])));
  hbMessage += 3;

  ASSIGN_AND_CREATE_MD_ENTRY(contents,hbMessage);

  if (hbtype == TLS1_HB_REQUEST) {
    unsigned char* response;
    ASSIGN_AND_CREATE_MD_ENTRY(response,
        CAST_UCHAR_PTR(
        MALLOC_WRAPPER(1 + 2 + payload + padding)));
    ...
    PUSH_TO_STACK(&response);
    PUSH_TO_STACK(&contents);
    MEMCPY_WRAPPER(response, contents, payload);
    ...
  }
  ...
}
```

Fig. 4. The instrumented version of the `process_heartbeat` function from Fig. 2.

functions that return pointers are replaced with calls to wrapper functions that handle the production of metadata information for the resulting pointers, as none is to be provided. The intended behavior of the original program is never altered by the addition of calls to the runtime monitoring library.

At the conclusion of the instrumentation phase, the AST is unparsed, giving us the instrumented source code file. After compiling and linking the output with the metadata libraries, we are left with an executable which can be run.

3.3. Runtime checking

The runtime monitoring framework of RTC is divided into four components: the metadata reference stack, the metadata entry table, the scope lock table, and the type history table. The function of these structures, depicted in Fig. 3, is to help detect spatial and temporal errors in the use of memory in the instrumented program. To check for spatial errors, RTC uses the metadata gathered about the pointer to determine whether its corresponding location falls within the lower and upper bounds of the memory allocation. If it should stray, an error will result. Meanwhile, RTC uses a "lock and key" approach to detect temporal memory errors. Every scope in the program has a lock and key associated with it. When a pointer is brought into existence in a given scope, the values of the lock and key are written to that pointer's metadata entry, and a lock entry is created in a separate data structure that attests that the pointer is valid for use in that scope. When a pointer is freed, the lock entry is removed, and the lock/key values in the pointer's metadata are erased. To check for a temporal error in the use of a pointer, the key value held in the pointer's metadata is compared with the key of the current scope's lock. If there is a mismatch between the actual and expected key values, it means that the memory that is pointed to has been freed and/or reallocated, and an error is thrown. At the end of the execution of the program, RTC expects that all locks have been relinquished, which means that if a call to `malloc` is not matched with a corresponding call to `free` before the end of the program, an error will be reported; in this regard, RTC is more demanding of the programmer than the C memory model.

Finally, we shall describe the runtime type-checking system which RTC provides. As we noted previously, RTC is capable of collecting type histories of pointers that can be checked against to detect possible runtime type violations. One important function of this system is to catch uses of variables that have not been properly initialized, which can be determined by checking flags associated with the type history, a constant-time operation. Having a comprehensive type history also allows RTC to detect latent type violations, such as when non-`void` pointers are passed to functions as `void` pointers, only to be recast on the other side as a type that is incompatible with the original. To catch these errors, RTC can audit a pointer's type history, checking each type in sequence to ensure type-correctness. Auditing the type history is a linear-time operation, but in practice this does not significantly add to RTC's overhead; checking the whole type history of a pointer is a rare occurrence, and type histories tend to be short (usually no more than three or four entries).

As an object lesson in how RTC exercises these powers, we can look at the example program. The key issue is that the `process_heartbeat` function relies on an implicit assumption, that the length of the user's message, contained in `hbMessage` equals the reported length (stored in `payload`), and the programmer made no provisions to guard against the violation of that assumption. If a malicious client overstates the size of their message to the server (e.g. sending a 1 byte message while claiming the message is 64 kilobytes in size), the server's call to `memcpy` will fill the buffer `response` with the original 1 byte message, followed by 63 980 bytes, taken from the contents of memory outside the bounds of `contents`.

When we enter the function `process_heartbeat`, RTC calls the function `CREATE_MD_ENTRY_IF_EXISTS` to check the stack address of the input string `hbMessage` against the address stored on the metadata stack. A metadata entry is created for the use of the `hbMessage` pointer in the current scope. Likewise, a separate metadata entry is created for the derived pointer `contents`. Finally, a metadata entry is created for the output buffer, `response`. When the program calls `MEMCPY_WRAPPER`, the wrapper function accesses the metadata associated with `contents` and `response`, to check that `payload` does not exceed the bounds of the allocation pointed to by `contents`. Then and only then can the call to `memcpy` proceed. If `memcpy` would read outside the bounds of `contents`, RTC raises an error and halts the execution.

3.4. Instrumentation reduction

Numerous empirical studies of software faults have noted that the frequency of bugs in software systems tend to follow the Pareto principle: 20% of software modules are responsible for 80% of software faults [10]. As a consequence, when using any run-time monitoring software, most safety checks will be performed on code that is bug-free, and this then implies that most overhead is unnecessary and potentially avoidable. There are, of course, very strong incentives for reducing any unnecessary instrumentation. For example, in the fields of embedded or network applications, software is often fine-tuned to deliver real-time responsiveness to external events, and, in the embedded case, the underlying architecture may place additional restrictions in terms of memory and power consumption. Being able to do in situ analysis of memory usage, where the software operates in a real-world environment and has to contend with real-world inputs, is an exciting prospect, but this requires that we have better, more fine-grained control over the overhead imposed by monitoring.

For this purpose, static analysis of the source code can be employed. Because RTC targets source code for instrumentation, and because it is built on top of a framework designed to facilitate static analysis, our tool is easily extended to support such analyses. After the initial traversal of the AST and the preprocessing phase, we are left with a list of program elements that are to be instrumented; a pass over that list is be made to exclude any program elements that are provably safe through static checking. However, we must ask what checks we can or should perform, and how frequently those

```
int f(int* arr, int sz) {
    ENTER_SCOPE();
    if(sz <= 0){
          ENTER_SCOPE();
          EXIT_SCOPE();
          return -1;
    }
    CREATE_MD_ENTRY_IF_EXISTS(&arr,GET_FROM_STACK(0));
    int tmp[2];
    ARRAY_BOUNDS_CHECK(sizeof(tmp) / sizeof(int), 0);
    tmp[0] = 0;
    ARRAY_BOUNDS_CHECK(sizeof(tmp) / sizeof(int), 1);
    tmp[1] = 1;


    for(int i = 0; i < sz; i++){
          ENTER_SCOPE();
          if( i % 2 == 0) {
               ARRAY_BOUNDS_CHECK(&arr,&arr[i]);
               ARRAY_BOUNDS_CHECK(sizeof(tmp)/sizeof(int),0);
               tmp[0] = (arr[i] > tmp[0] ? arr[i] : tmp[0]);
          } else {
               ARRAY_BOUNDS_CHECK(&arr,&arr[i]);
               ARRAY_BOUNDS_CHECK(sizeof(tmp)/sizeof(int),1);
               tmp[0] = (arr[i] > tmp[0] ? arr[i] : tmp[0]);
          }
          ARRAY_BOUNDS_CHECK(sizeof(tmp)/sizeof(int),i%2);
          tmp[i%2] = (arr[i] > tmp[i%2] ? arr[i] : tmp[i%2])
          EXIT_SCOPE();
    }

    ARRAY_BOUNDS_CHECK(sizeof(tmp) / sizeof(int), 0);
    ARRAY_BOUNDS_CHECK(sizeof(tmp) / sizeof(int), 1);
    EXIT_SCOPE();
    return (tmp[0] > tmp[1] ? 0 : 1);
}
```

Fig. 5. An example of where instrumentation reduction can be used to improve performance.

checks will reveal that code is in fact safe. That then is the subject of this subsection. As an example, in Fig. 5, we give an arbitrary program derived from one generated by Csmith, a randomized test-case generation tool for C compilers [35].

3.4.1. Checking allocation bounds

The first and most promising targets for instrumentation reduction are allocation bounds checks. If an allocation is stack-based or is allocated dynamically with constant size, or if the access has a constant index or a constant-bounded variable index, then we may be able to either eliminate the bounds check or condense a series of bounds checks. This process can be accomplished through the use of expression evaluation and loop invariant code motion. For a comprehensive look at bounds checking elimination, we recommend Würthinger et al. 2009, who considered the problem in the context of the Java language [33].

In Fig. 5, we have two allocations that are being tracked, the first being `tmp`, a stack allocation, and `arr`, a heap allocation. Both are subject to many reads and/or writes and this causes the creation of bounds checks to protect those accesses. Many, however, can be eliminated safely.

First, for the bounds checks on lines 10 and 12, guarding the initialization of `tmp`, constant evaluation gives us the indices 0 and 1 respectively, and, given that we can pull the length of `tmp` from its declaration, we can conclude these indices respect allocation bounds, implying that the operations are safe and that the bounds checks can be eliminated. By the same logic, we are able to eliminate the checks on lines 20, 25, 32, and 33.

Next, we can consider the checks guarding accesses to `arr` within the bounds of the `for` loop on lines 20 and 23. We observe that in both cases the index is computed based on the loop index variable, and because the surrounding loop is in canonical form, ROSE allows us to easily extract its upper and lower bounds, namely $i=0$ and $i=n-1$. Having this knowledge, we can replace each of these checks with two checks, placed before the loop, on its lower and upper bounds; this has the effect of eliminating checks on indices from $i=1$ to $i=n-2$. Furthermore, because checks at lines 20 and 23 are identical, we can eliminate one in the process, leaving us with two new checks instead of four.

There are, however, several limitations with this approach. It is possible, for instance, that a pointer to `arr` could be passed to a function `f` within the loop, and a call to `free` is made within that function, deallocating it. Since we cannot see or predict the side effects of `f`, we cannot do away with the metadata lookups that occur as part of the bounds checks, or we would risk missing a temporal violation.

3.4.2. Eliminating scope tracking

That having been said, it is often the case that a temporal violation is rendered impossible or would have been caught by checks moved outside of the loop, in which case we can eliminate unnecessary scope tracking operations. It is already the case that RTC is overzealous with adding scope tracking operations because these are put in place before other instrumentations that may depend upon them, as we can see on lines 4 and 5. Likewise, after an initial pass has been made to condense or eliminate unnecessary checks, the scope tracking operations can usually be excised. For this, a second pass can be made over the list of instrumentation candidates, and if the subtree underneath a scope contains no candidates for instrumentation, then the scope itself can be removed from consideration. This allows us to remove the `ENTER_SCOPE` and `EXIT_SCOPE` calls on lines 17 and 29, which would otherwise be called a total of 2 ∗ `sz` times during the execution of the loop.

3.4.3. Metadata lookups

Another source of inefficiency that can be eliminated through analysis are redundant lookups for the same metadata. For example, the two bounds checks created for `arr` would each perform a separate lookup for the metadata associated with that heap allocation. When two or more lookups involving the same alias occur within the same scope, it is possible for us to create a call that performs the lookup once and then passes the resulting metadata struct to the bounds checking functions. However, careful analysis is required to check for redefinitions of the alias, as RTC will create new metadata for that redefinition.

3.4.4. Remarks

It is worth noting that several alternatives were considered as complementary solutions to the problem of overhead reduction, and we wish to mention them here.

One option is to allow the user disable whole classes of checks in order to save on overhead. In some respects, this would be in keeping with RTC's emphasis on the selective instrumentation of modules. However, within a large, real-world C program, all classes of operations that RTC is capable of instrumenting can be found in abundance, and disabling classes of checks based on intuition or guesswork invites the potential for false negatives. Furthermore, disabling instrumentation can affect the quality of the results that RTC provides. For example, a check may reveal an error, but an earlier check that was disabled, one closer to the root cause of the problem, would have provided more revealing information.

Another option is to lift various assertions and routines found in the run-time library into the instrumented source code so as to expose them to the compiler. For example, it is possible that there can be two or more identical checks on the bounds of the same allocation, and the underlying assertions could be eliminated by compiler optimization, but this is not possible because the assertions are hidden behind library calls. At the same time, while this would be beneficial for eliminating checks on stack allocations, it would be less helpful for heap allocations, because checks on heap allocations require metadata lookups which can not be automatically optimized away at compile-time.

The last and perhaps most interesting consideration would be to reduce overhead by staggering instrumentations across multiple versions of a function, and to use the original function as a wrapper that routes callers to these different versions of the code non-deterministically. For network applications, where throughput is paramount and bugs such as Heartbleed take place over the course of thousands of calls, one could thin the spread of instrumentation as needed to control overhead without significantly harming bug detection capabilities. Meanwhile, for the embedded case, different versions of the instrumented code could be deployed to different devices, minimizing the burden on any one device while still providing suitable protection for the population as a whole. However, this solution implies a risk of false negatives: if a check were to be deployed in just one out of N versions of the code, and that check would be necessary to detect a bug, the bug will go undetected in $(N-1)/N$ calls. On the other hand, if we were to increase the frequency of the check, we would lose the benefits of staggering.

4. Evaluation

4.1. Estimating overhead costs

We collected a handful of C implementations of common algorithms to help give a sense of the overhead costs incurred by RTC. All implementations were single thread. The following is a list of the algorithms that were tested:

1. Binary search: C implementation of the binary search algorithm. Searches a sorted array of elements in the hopes of finding a target value.
2. DTW: Dynamic time warping algorithm. Takes two input files containing numerical sequences and computes a measure of similarity between them.
3. Heapsort: Sort a list according to the heapsort algorithm. This is an in-place implementation.
4. MD5: A severely compromised cryptographic hash function.
5. Mergesort: Sort a list according to the mergesort algorithm.

Each of these algorithms was tested with a reasonable size of input and the execution times of the uninstrumented and instrumented implementations was compared. The experiments on an Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60 GHz, running the CentOS 7 operating system, and all code was compiled with version 4.8.2 of the GCC compiler with only default optimizations enabled. Each experiment was performed three times, and the average of these results was recorded. The results of these experiments can be seen in Fig. 6. For the sake of comparison, we also provide results for Clang's AddressSanitizer and Valgrind's Memcheck which we discuss in greater detail in Section 4.3. These tools were chosen because they are popular, well-supported, and offer similar capabilities. We note that underflow and overflow detection was disabled for RTC and use-after-return detection was disabled for AddressSanitizer.

In the case of the binary search implementation, the bulk of the overhead comes from the bounds check added to every array access. For an input of N elements, binary search will require at most $\lfloor \log_2(N)+1 \rfloor$ iterations of the search, which means that there will also be that many calls to the array bounds checking function. A breakdown of the calls made to the metadata libraries reveals that these checks account for 97% of all calls. Many of these calls can be condensed and moved out of loop bodies, as we described previously, resulting in a 23% reduction in overhead.

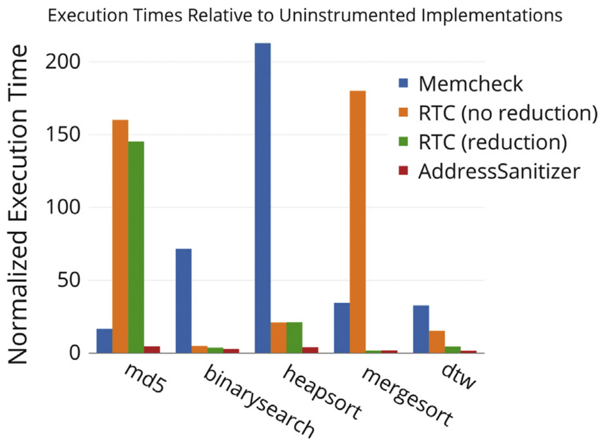

Fig. 6. Normalized execution times for a selection of C implementations of algorithms instrumented by RTC, Clang's AddressSanitizer, and Valgrind's Memcheck.

Next, comparing the results of the mergesort and heapsort implementations can show us the cost incurred by RTC's pointer monitoring facilities. Both mergesort and heapsort exhibit $O(N\log N)$ worst case performance, and we can expect the worst case cost in overhead for every additional element to be commensurate with this figure. As in the binary search implementation, both the mergesort and heapsort operations are dominated by array access operations in comparable measure. We note that the mergesort implementation does use more memory than the heapsort implementation, but the only difference this makes in terms of the cost of the instrumentation is that an additional entry is made in the metadata table for every new temporary array, a relatively inexpensive operation. This is to say that the quantity of memory that is allocated by a program does not directly affect the overhead costs, only the extent to which that memory is used. However, we do note both implementations pass pointers to the functions they call, and all operations involving pointers are mediated by specialized structs and calls to the metadata library, and these can significantly add to the overhead costs. However, this also creates many opportunities for our static analysis routines to eliminate and condense checks. For mergesort, reduction eliminates 98.42% of all scope tracking operations, and 99.99% of bounds checking operations (both those requiring metadata lookups and not), bringing the execution time down from roughly 40 s to 0.4 s, causing RTC to ever-so-slightly outperform AddressSanitizer. We were not, however, able to make any improvements on the performance of RTC on the heapsort benchmark. All of the operations performed on the data structure are hidden behind helper functions which inhibit our intraprocedural static analysis from easily identifying when instrumentation can be eliminated safely.

This then brings us to the MD5 and DTW cases, which are representative of the kinds of real-world code that we are targeting with RTC. In both cases, the original source codes are awash with pointers, from calls to `malloc`, `realloc`, and `free`, to pointer dereferencing and type casting operations. Ensuring that all pointers are being used safely and correctly is no small task. However, we do note that instrumentation reduction eliminates roughly 10% of overhead on the MD5 benchmark and 70% of overhead on the DTW benchmark, and in the latter case RTC's performance is brought within close range of AddressSanitizer (4.47 × execution time overhead for RTC vs. 1.47 × for AddressSanitizer). For the MD5 benchmark, our attempt to eliminate checks within the main loop is limited by the fact that the expressions used to compute the indices for the left rotation can vary depending on the current value of the loop index (e.g. $index==i$ if $i<=15, index==(5i+1) \bmod 16$ if $16<=i<=31$). Overall, while RTC's execution overhead is slightly higher on average than AddressSanitizer, it remains within tolerable limits.

Next, in Fig. 7, we see that has the smallest memory footprint of any of the tools, often beating the competition by a factor of 10–90. The only benchmark where a tool has slightly lower memory consumption is the mergesort benchmark, due to the fact that we must track an abundance of temporary arrays and RTC delays freeing allocations for metadata, which pushes the peak resident set size slightly over that of AddressSanitizer. This then is one of the strongest selling points of RTC.

4.2. Code coverage

To assess the bug coverage of RTC, we opted to test our tool against the Stonesoup test suite. Stonesoup refers to a curated test suite for C and Java published by the NIST in 2012, containing 460 cases dealing with issues of memory corruption and null pointer dereference. Each test case is well-documented, which means that we were able to instrument RTC to see whether it could detect the specific bug given in the description of the test case.

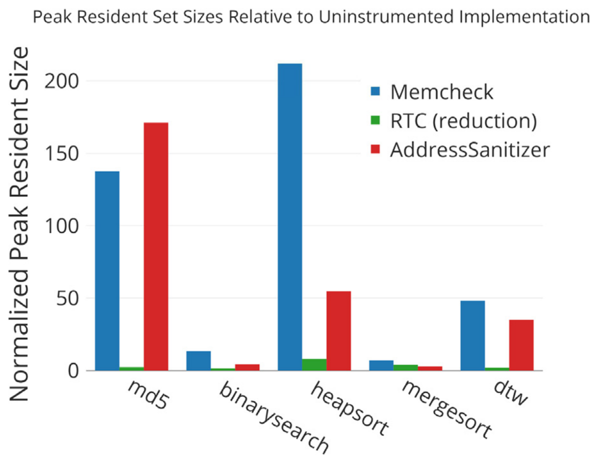

Fig. 7. Normalized maximum resident set sizes for a selection of C implementations of algorithms instrumented by RTC, Clang's AddressSanitizer, and Valgrind's Memcheck. Note that because our instrumentation reduction process does not eliminate calls for metadata creation and elimination, reduction has no impact on memory footprint (though this may change in future versions).

4.2.1. *Palindrome finder*

One subcategory of the memory corruption test suite consists of variants of a program that takes a palindromic number as input and computes the next palindromic number. In each of these programs, a heap-based buffer overflow has been placed into the code that can occur when reading in the number to find the next palindrome. An attacker can provide input of a size larger than the intended maximum, and use this to launch a buffer overflow attack. Out of the 15 test cases, four of them used the X11 library to produce a GUI for the user to provide input, and these were excluded from consideration. Out of the remaining eleven test cases, RTC succeeded at detecting bugs in nine. On the same test cases, Valgrind's Memcheck succeeded at detecting bugs in only five out of the ten. Clang's AddressSanitizer, meanwhile, aborted execution on six of the ten cases, having detected a segfault but being unable to explain why it occurred, and failed to find any bugs in the remaining four.

4.2.2. *Solitaire cipher*

Another subcategory of the memory corruption test suite contains 20 variants of a solitaire encryption cipher program. In each case, the program was modified to include a buffer overflow vulnerability when reading in the key file that is used to create the random seed for shuffling. Five of them used the X11 library to handle user input, and these were excluded from consideration. Out of the remaining 15 test cases, RTC succeeded in finding bugs in 6 of them. RTC failed to produce compilable code for the remaining cases, due to the a mishandling of an unusual array declaration. This indicates that more work is needed to ensure that RTC handles all of the corner cases of the C language. For the sake of comparison, Memcheck succeeded in finding bugs in 13 out of the 15 cases. AddressSanitizer aborted on 5 cases (for the same reason as in the palindrome tests), failed to detect any bugs on 6 cases, and successfully detected bugs on 4 cases.

4.3. *Comparison of RTC to similar tools*

Now we shall compare and contrast RTC to two other contemporary tools, Valgrind's Memcheck and Clang's Address-Sanitizer. A summary of the points covered in this subsection can be found in Table 1.

4.3.1. *Target of instrumentation*

Both RTC and AddressSanitizer target source code for instrumentation; Memcheck targets binaries. The advantage for Memcheck is that it is completely language independent. However, working from the binary rather than the original source code means that type information is lost. One consequence is that invalid memory is reported when observable behavior is affected, but not when invalid memory is read. The lack of type information demands this behavioral definition, as compilers frequently emit load instructions for padding memory (e.g. a short value in a struct is aligned on word boundaries, and frequently compilers load the entire word instead of the smaller short). RTC and AddressSanitizer, meanwhile, can instrument all pointer accesses, which allows both tools to report uses of invalid memory when they occur. We also note that RTC adds its instrumentation prior to compiler optimization, whereas Memcheck and AddressSanitizer do so after compiler optimizations are complete [17].

4.3.2. *Use of shadow memory*

Both Memcheck and AddressSanitizer track shadow allocations of memory in order to determine when and where those allocations are valid for use. Valgrind's Memcheck tracks the validity of heap allocations on a per bit level, that is, one check bit for every bit of allocated memory. Clang's AddressSanitizer uses one bit of shadow memory per byte of real memory. With both tools, whenever memory is freed, that memory is marked as invalid for use and is rendered inaccessible. In contrast, RTC shadows pointers, rather than allocations; the size of an allocation does not add to the amount of shadow memory required to track its use.

On a 64-bit architecture, a local metadata struct requires 128 bits of memory: 64 to hold the pointer itself and another 64 to hold the stack address of the pointer. The metadata entry, meanwhile, requires another 256 bits of memory: 128 for the

Table 1
Comparison of RTC to other contemporary tools. Note that AddressSanitizer's support for use-after-return and memory leak detection are experimental, and use-after return detection is disabled by default.

Feature	Mem-check	Address-Sanitizer	RTC
Target	Binary	Source code	Source code
Ratio of shadow memory to real memory	1:1	1:8	N/A
Average slowdown	22 × [24]	2 × [32]	9.7 × (geometric), 35.3 (arithmetic)
Out-of-bounds accesses (stack, heap, global)	No (heap only)	Yes	Yes
Use of uninitialized values	Yes	Yes	Yes
Invalid/double free	Yes	Yes	Yes
Use-after-free	Yes	Yes	Yes
Use-after-return	No	Yes	Yes
Memory leaks	Yes	Yes	Yes
Arithmetic overflows/underflows	No	Yes	Yes

lock and key associated with the scope of the pointer, and 128 to record both the lower and upper bounds of the allocation. Finally, an individual typetable entry consists of a variable-length sequence of 64-bit pointers to typeinfo nodes, and a pointer must have at least one type associated with it; this brings us to a minimum of 448 bits of information per pointer. Considering that the number of pointers and the size of allocations can vary from one program to the next, the relative size of RTC's shadow memory footprint can be difficult to estimate. However, on average, the number of pointers in use in a program at any one given time, multiplied by 448, is smaller than the quantity of allocated memory.

4.3.3. Execution overhead

On average, AddressSanitizer produces the lowest overhead of the three tools, followed by RTC and then by Memcheck. In practice, the actual overhead experienced depends heavily on the qualities of the program being instrumented. Because Memcheck requires very fine-grained monitoring, it is heavily affected by the size and quantity of memory allocations in addition to the volume of reads and writes from and to those allocations. AddressSanitizer employs a similar scheme but at one-eighth of the resolution, which, in conjunction with information derived from the source code (e.g. type information), reduces the costs associated with monitoring. Because RTC foregoes direct monitoring, its performance hinges upon the number of pointers used and the number of interactions with them, which we noted in our discussion of the findings seen in Fig. 6.

4.3.4. Bug detection capabilities

Unlike Memcheck but on par with AddressSanitizer, RTC is capable of tracking uses of memory on both the stack and the heap and has the ability to catch arithmetic underflow and overflow errors. Compared to AddressSanitizer, RTC offers the same features with the addition of support for run-time type-checking.

4.3.5. Remarks

While Valgrind's Memcheck is and will remain a staple tool for traditional computing platforms, it is ill-suited for analysis of small and embedded systems. The creators of the tool have made clear that the maintenance burden is too great to offer broad support for embedded environments. Even if that were not the case, targeting binaries inhibits selective instrumentation and the loss of high-level type information means that the tool cannot perform the kinds of reductions that RTC can, resulting in an unavoidable excess of execution overhead. That overhead makes Memcheck a poor choice for in situ analysis of always-on, performance-driven applications like network systems.

AddressSanitizer, meanwhile, has much in common with our tool, but there are key differences that make RTC a better choice for the kinds of applications and environments that we are targeting. Like Valgrind, AddressSanitizer requires shadow memory, which imposes memory overhead that is difficult to manage in environments where storage is a luxury. In contrast, RTC decouples its storage requirements from the amount of memory used by an application; the memory needs of RTC are the bare minimum necessary to be able to identify violations. However, RTC does trade space for time, which gives AddressSanitizer an advantage in terms of execution overhead, although the gap is narrowed considerably through optimizations and instrumentation reduction. In practice, however, this is less of a concern, in part because it is possible to stagger instrumentation across multiple implementations, as was discussed previously.

5. Conclusion and future work

In this paper, we have presented RTC, a source code instrumentation tool that finds software flaws common when using the C programming language. Our tool is capable of handling real-world programs. In our tests, we have demonstrated that RTC finds most bugs in available error detection benchmarks suits. Future research directions include extending RTC to handle multithreaded C code, and to incorporate a probabilistic model of errors to determine the best policy for staggering instrumentation across versions of functions, in order to bring overhead down even further while not compromising bug detection coverage.

References

[1] Austin TM, Breach SE, Sohi GS. Efficient detection of all pointer and array access errors. SIGPLAN Not 1994;29(June (6)):290–301.

[2] Burnim J, Elmas T, Necula G, Sen K. NDSeq: runtime checking for nondeterministic sequential specifications of parallel correctness. In: PLDI '11: proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation. ACM; June 2011.

[3] Burrows M, Freund SN, Wiener JL. Run-time type checking for binary programs. In: Proceedings of the 12th international conference on compiler construction, CC'03. Berlin, Heidelberg: Springer-Verlag; 2003. p. 90–105.

[4] Chen S, Kozuch M, Strigkos T, Falsafi B, Gibbons PB, Mowry TC, et al. Flexible hardware acceleration for instruction-grain program monitoring. In: 35th international symposium on computer architecture, 2008, ISCA '08. 2008. p. 377–88.

[5] Delahaye M, Kosmatov N, Signoles J. Common specification language for static and dynamic analysis of c programs. In: Proceedings of the 28th annual ACM symposium on applied computing. ACM; 2013. p. 1230–5.

[6] Devietti J, Blundell C, Martin MMK, Zdancewic S. Hardbound: architectural support for spatial safety of the c programming language. In: Proceedings of the 13th international conference on architectural support for programming languages and operating systems, ASPLOS XIII. New York, NY, USA: ACM; 2008. p. 103–14.

[7] El-Ghazawi T, Carlson W, Sterling T, Yelick K. UPC distributed shared memory programming. Wiley series on parallel and distributed computing, 1st ed. Wiley; 2003.
[8] Falsafi B, Gibbons PB, Kozuch M, Mowry TC. Log-based architectures for general-purpose monitoring of deployed code. In: Proceedings of the 1st workshop on architectural and system support for improving software dependability, 2006.
[9] Goodstein ML, Vlachos E, Chen S, Gibbons PB, Kozuch MA, Mowry TC. Butterfly analysis: adapting dataflow analysis to dynamic parallel monitoring. In: ASPLOS XV: proceedings of the fifteenth edition of ASPLOS on architectural support for programming languages and operating systems. ACM; March 2010.
[10] Grbac TG, Huljenić D. On the probability distribution of faults in complex software systems. Inf Softw Technol 2015;58:250–8.
[11] Grimmer M, Schatz R, Seaton C, Würthinger T, Mössenböck H. Memory-safe execution of c on a java vm. In: Proceedings of the 10th ACM workshop on programming languages and analysis for security. ACM; 2015. p. 16–27.
[12] IBM: Rational PurifyPlus family ⟨http://ibm.com/software/products/en/purifyplus⟩, 2014, accessed on March 12, 2014.
[13] Jakobsson A, Kosmatov N, Signoles J. Fast as a shadow, expressive as a tree: hybrid memory monitoring for c. In: Symposium on applied computing (SAC'15). À parai\widehattre, 2015.
[14] Jim T, Morrisett JG, Grossman D, Hicks MW, Cheney J, Wang Y. Cyclone: a safe dialect of c. In: Proceedings of the general track of the annual conference on USENIX annual technical conference, ATEC '02. Berkeley, CA, USA: USENIX Association; 2002. p. 275–88.
[15] Kalajdzic K, Bartocci E, Smolka SA, Stoller SD, Grosu R. Runtime verification with particle filtering. In: Legay A, Bensalem S, editors. 4th international conference on runtime verification (RV'13). Lecture notes in computer science, vol. 8174. Berlin, Heidelberg: Springer; 2013. p. 149–66.
[16] Kosmatov N, Petiot G, Signoles J. An optimized memory monitoring for runtime assertion checking of c programs. In: Legay A, Bensalem S, editors. 4th international conference on runtime verification (RV'13). Lecture notes in computer science, vol. 8174. Berlin, Heidelberg: Springer; 2013. p. 167–82.
[17] Liu T, Tian C, Hu Z, Berger ED. Predator: predictive false sharing detection. In: ACM SIGPLAN notices, vol. 49. ACM; 2014. p. 3–14.
[18] Luecke GR, Coyle J, Hoekstra J, Kraeva M, Xu Y, Park MY, et al. The importance of run-time error detection. In: Parallel tools workshop, 2009. p. 145–55.
[19] Mekkat V, Holey A, Zhai A. Accelerating data race detection utilizing on-chip data-parallel cores. In: Legay A, Bensalem S, editors. 4th international conference on runtime verification (RV'13). Lecture notes in computer science, vol. 8174. Berlin, Heidelberg: Springer; 2013. p. 201–18.
[20] Muzahid A, Gracia DS, Qi S, Torrellas J. SigRace: signature-based data race detection. In: ISCA, 2009. p. 337–48.
[21] Nagarakatte S, Martin MMK, Zdancewic S. WatchdogLite: hardware-accelerated compiler-based pointer checking. In: CGO '14: proceedings of annual IEEE/ACM international symposium on code generation and optimization. ACM; February 2014.
[22] Nagarakatte S, Zhao J, Martin MMK, Zdancewic S. Softbound: highly compatible and complete spatial memory safety for c. In: Proceedings of the 2009 ACM SIGPLAN conference on programming language design and implementation, PLDI '09. New York, NY, USA: ACM; 2009. p. 245–58.
[23] Necula GC, Condit J, Harren M, McPeak S, Weimer W. Ccured: type-safe retrofitting of legacy software. ACM Trans Program Lang Syst 2005;27(May (3)):477–526.
[24] Nethercote N, Seward J. How to shadow every byte of memory used by a program. In: Proceedings of the 3rd international conference on virtual execution environments, VEE '07. New York, NY, USA: ACM; 2007. p. 65–74.
[25] Nethercote N, Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not 2007;42(June (6)):89–100.
[26] Norris B, Demsky B. CDSchecker: checking concurrent data structures written with C/C++ atomics. SIGPLAN Not 2013;48(October (10)):131–50.
[27] Parasoft Inc.: Insure++ ⟨http://www.parasoft.com/insure⟩, 2014, accessed on March 12, 2014.
[28] Pirkelbauer P, Liao C, Panas T, Quinlan D. Runtime detection of C-style errors in UPC code. In: 5th conference on partitioned global address space models (PGAS), Galveston, TX, 2011.
[29] SANS Institute: CWE/SANS TOP 25 most dangerous software errors ⟨http://www.sans.org/top25-software-errors⟩, 2011.
[30] Simpson MS, Barua RK. Memsafe: ensuring the spatial and temporal memory safety of c at runtime. Softw: Pract Exp 2013;43(1):93–128.
[31] Tassey G. The economic impacts of inadequate infrastructure for software testing. NIST report 02-3, 2002.
[32] The Clang Team ⟨http://clang.llvm.org/docs/AddressSanitizer.html⟩, September 2014.
[33] Würthinger T, Wimmer C, Mössenböck H. Array bounds check elimination in the context of deoptimization. Sci Comput Program 2009;74(5–6):279–95. [special issue on Principles and Practices of Programming in Java (PPPJ 2007)].
[34] Xu W, DuVarney DC, Sekar R. An efficient and backwards-compatible transformation to ensure memory safety of c programs. In: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on foundations of software engineering, SIGSOFT '04/FSE-12. New York, NY, USA: ACM; 2004. p. 117–26.
[35] Yang X, Chen Y, Eide E, Regehr J. Finding and understanding bugs in c compilers. In: ACM SIGPLAN notices, vol. 46. ACM; 2011. p. 283–94.
[36] Zhang W, Lim J, Olichandran R, Scherpelz J, Jin G, Lu S, et al. Conseq: detecting concurrency bugs through sequential errors. In: Proceedings of the sixteenth international conference on architectural support for programming languages and operating systems, ASPLOS XVI. New York, NY, USA: ACM; 2011. p. 251–64.
[37] Zhou P, Qin F, Liu W, Zhou Y, Torrellas J. iWatcher: efficient architectural support for software debugging. In: 31st annual international symposium on computer architecture, 2004, Proceedings, 2004. p. 224–35.