View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Dayton

University of Dayton
eCommons

Computer Science Faculty Publications

Department of Computer Science

8-2016

Leveraging Static Analysis Tools for Improving Usability of Memory Error Sanitization Compilers

Rigel Gjomemo
University of Illinois at Chicago

Phu Huu Phung
University of Dayton, pphung1@udayton.edu

Edmund Ballou
Nokia Bell Labs

Kedar S. Namjoshi
Nokia Bell Labs

V. N. Venkatakrishnan
University of Illinois at Chicago

See next page for additional authors

Follow this and additional works at: https://ecommons.udayton.edu/cps_fac_pub

Part of the Graphics and Human Computer Interfaces Commons, and the Other Computer Sciences Commons

eCommons Citation

Gjomemo, Rigel; Phung, Phu Huu; Ballou, Edmund; Namjoshi, Kedar S.; Venkatakrishnan, V. N.; and Zuck, Lenore, "Leveraging Static Analysis Tools for Improving Usability of Memory Error Sanitization Compilers" (2016). *Computer Science Faculty Publications*. 141.
https://ecommons.udayton.edu/cps_fac_pub/141

This Conference Paper is brought to you for free and open access by the Department of Computer Science at eCommons. It has been accepted for inclusion in Computer Science Faculty Publications by an authorized administrator of eCommons. For more information, please contact frice1@udayton.edu, mschlangen1@udayton.edu.

Author(s)
Rigel Gjomemo, Phu Huu Phung, Edmund Ballou, Kedar S. Namjoshi, V. N. Venkatakrishnan, and Lenore Zuck

This conference paper is available at eCommons: https://ecommons.udayton.edu/cps_fac_pub/141

Leveraging Static Analysis Tools for Improving Usability of Memory Error Sanitization Compilers

Rigel Gjomemo*, Phu H. Phung†, Edmund Ballou*, Kedar S. Namjoshi‡, V.N. Venkatakrishnan*, and Lenore Zuck*
*University of Illinois, Chicago, USA
Email: rgjomeme1,eballou,venkat@uic.edu/lenore@cs.uic.edu
†University of Dayton, USA
Email: phu@udayton.edu
‡Bell Labs, Nokia
Email: kedar@research.bell-labs.com

***Abstract*—Memory errors such as buffer overruns are notorious security vulnerabilities. There has been considerable interest in having a compiler to ensure the safety of compiled code either through static verification or through instrumented runtime checks. While certifying compilation has shown much promise, it has not been practical, leaving code instrumentation as the next best strategy for compilation. We term such compilers Memory Error Sanitization Compilers (MESCs). MESCs are available as part of GCC, LLVM and MSVC suites. Due to practical limitations, MESCs typically apply instrumentation indiscriminately to every memory access, and are consequently prohibitively expensive and practical to only small code bases. This work proposes a methodology that applies state-of-the-art static analysis techniques to eliminate unnecessary runtime checks, resulting in more efficient and scalable defenses. The methodology was implemented on LLVMs Safecode, Integer Overflow, and Address Sanitizer passes, using static analysis of Frama-C and Codesurfer. The benchmarks demonstrate an improvement in runtime performance that makes incorporation of runtime checks a viable option for defenses.**

I. Introduction

Security vulnerabilities resulting from unsafe memory accesses such as buffer overruns are notorious. Starting from the highly publicized Morris worm in the 80s, exploits resulting from memory safety errors have received considerable attention. Prevention of these errors has been a topic of intense research over the past two decades.

While manual methods are currently the most widely adopted approach, they are either error prone or tedious for large applications as well as legacy code. We thus focus here on automated approaches.

An attractive automated method for vulnerability protection is to use a compiler for ensuring safety of the code it produces. Code that passes through a compiler can be checked or retrofitted with defenses, and the approach can be transparently applied to large codebases. In particular, in several approaches, the compiler is used to instrument the compiled code with runtime checks that ensure the safety of memory related accesses. We term such compilers Memory Error Sanitization Compilers (MESCs). MESCs have been developed and available as part of compiler suites such as GCC (as of gcc 4.8)[1], LLVM framework (from Clang compiler version 3.1)[2] and MSVC (Microsoft Visual Studio platforms). Since every piece of code that is compiled goes through a compiler toolchain, it is possible for this approach to be transparently applied to codebases that use the compiler toolchain. Compared to stand-alone tools [5] for retrofitting code, a compiler-based retrofitting strategy has a better chance of critical mass adoption.

A main challenge with code that is retrofitted for memory safety is performance. While there has been considerable research in the recent past to address this issue, much of this work has not become part of MESCs. One reason is that compiler writers hesitate to include analysis algorithms of high complexity in the compiler toolchain for reasons of (static) compile-time performance. The GCC wiki[3] has as rule 1: "Do not add algorithms with quadratic or worse behavior, ever."

Due to the lack of high-precision algorithms for performing a precise instrumentation, MESCs typically apply instrumentation indiscriminately to every memory access. (Section II has a brief analysis of the performance of LLVM's instrumentation). Hence, they do not scale, resulting in prohibitive overhead.

This work proposes a methodology to curb the performance costs of software compiled with a MESC. Our approach is to facilitate the use of state-of-the-art static analysis techniques by incorporating their results inside the compiler to eliminate unnecessary runtime checks, making this class of defenses more efficient and scalable. Our main contribution is to build a practical linkage between LLVM and static analysis tools and use this to reduce sanitization overhead. The main benefits of our approach are:

- *No compiler modification for analysis:* Our approach utilizes the results of state-of-art analysis algorithms inside the compiler, without changing the compiler analysis procedures. (Our approach does require modifications to the retrofitting module of the compiler in order to make use of these results.)
- *Generality:* Our approach is general enough to include the

[1] https://gcc.gnu.org/
[2] http://llvm.org/
[3] https://gcc.gnu.org/wiki/Speedup_areas

978-1-5090-4127-5/16 $31.00 © 2016 IEEE
DOI 10.1109/QRS.2016.44

results of any static analysis tool. In our implementation, we have utilized the results of Frama-C and CodeSurfer, two of state-of-the-art static analysis tools.
- *Performance:* Our approach has been tested with a variety of benchmarks, including small and large applications. These benchmarks demonstrate improvements in runtime performance that make incorporation of runtime checks a viable option for defenses.

This paper is organized as follows: Section II motivates the work, describing the need for runtime checks and the shortcomings of current implementations. Section III provides the high-level architecture of the design. Section IV details the implementation. Section V presents data from our experiments using the method. Section VI describes previously published work that address these issues. Conclusions are given in Section VII.

II. Background and Problem Statement

Memory safety related errors constitute some of the most critical security bugs in programs. There is a long history of security incidents whose root-cause is due to errors such as out-of-bounds access, integer overflow, and use-after-free scenarios.

There is also a long history of security defenses for these types of attacks, a focus of intense research over two decades. Our focus here is on directly addressing vulnerabilities; mitigation defenses are discussed in Section VI.

Despite the intense level of focus, software vulnerabilities abound. We discuss the main issues that contribute to this next.
Issue 1: Performance. The overhead of runtime checking that is required to prevent memory safety errors has been high (overheads from 2x to 80x). There has been considerable progress made with respect to performance and we can hope that this trend will lead to efficient defenses. However, given that a wide range of software is developed and distributed, in order to achieve critical mass, they need to be hardened through developer-transparent toolchains. Compiler writers have recognized this need, and have integrated memory error sanitization techniques in the compilation cycle. As mentioned earlier, we call these compilers Memory Error Sanitization Compilers (MESCs). MESCs are an attractive solution to the critical mass adoption problem and have been developed for mainstream compiler suites, *e.g.,* GCC, LLVM and MSVC. While these sanitization passes are rarely incorporated in production software due to performance concerns, their functionality aids in testing, error detection and error diagnosis.
Issue 2: Precision. Practical (compile-time) performance requirements on a production compiler do not facilitate using advanced analysis techniques (for instance, using quadratic or even super-linear algorithms). This limits the precision of the analysis results and, in turn, the optimizations which can be performed. Secondly, it requires non-trivial effort to build compiler passes that incorporate algorithms yielding more precise results. As a result of these two factors, end users of MESCs do not benefit from the recent advances in static analysis algorithms that could improve the runtime overheads due to instrumentation.
Issue 3: Lack of critical mass adoption. Many defense techniques developed have been through stand-alone implementations or ad-hoc extensions of existing compilers. For a defense technique to become mainstream, critical mass adoption in a developer-transparent toolchain framework is essential.

A. Analysis of runtime overheads

In a preliminary work we assessed candidate benchmarks for testing the efficacy and usability of three sanitization protocols, measuring the performance overhead imposed on the applications by the inserted runtime checks [6]. The three sanitization tools, Safecode [7], Address Sanitizer (ASAN) [8], and Integer Overflow Checks (IOC) [9], are discussed in detail below. Similar conditions obtained as for the measurement runs reported in the Evaluation section (Section V). Our runtime data are shown in Table I, which displays the overhead with respect to the original introduced by the different sanitizations.

We find large overhead for most of the benchmarks, with some exhibiting a slowdown exceeding a factor of 90 times slower than the original code; a few showed modest performance cost, as low as 9%. These costs present a challenge to the security community, for runtime enhancements to become acceptable in production software. We note that for Safecode, throughout our experiments, we used the production version available at the official page [4], which, according to the developers does not incorporate certain unstable optimizations. The current work addresses this situation by designing strategies for targeted restriction of runtime checks insertion.

Benchmark	Safecode	IOC	ASAN
oggenc	0.28	0.21	3.48
LasPack	30.30	0.97	4.29
gzip	15.70	0.22	0.94
debie1	46.12	0.56	4.46
appbt	97.98	4.85	2.60
bzip2	70.15	0.39	3.87
susan	18.23	2.35	4.12
quicklz	19.04	0.59	1.80
cpumaxmp64	4.00	0.09	0.07
linpack	28.00	0.34	3.44
NEC-Matrix	55.67	18.8	4.63

TABLE I: Benchmark Overhead due to Runtime Checks, for Three Sanitization Tools

B. Optimizing runtime checks

To address *Issue 1*, current implementations of runtime checks in MESC deal only with reducing the overhead of the runtime infrastructure through a variety of implementation strategies that involve the runtime data-structures (e.g., fat pointers [7], [10], shadow memory [8], pool allocation [11]).

[4]http://safecode.cs.illinois.edu/

Our solution to *Issue 1* is to reduce the time overheads due to runtime checks by making use of precise static analysis. The results from such analysis are then used to remove those checks that can be statically determined to be safe. Indeed, every MESC employs several static analysis algorithms to perform optimizations, but production compilers typically restrict these algorithms to the most efficient ones, not necessarily the most precise.

To address *Issue 2*, we present a method to improve the time performance of MESC compiled code by leveraging *external static analysis* tools. We aim to develop an approach that has the same safety guarantees of a conventional MESC without the runtime overheads, specifically retaining memory safety while removing unnecessary checks. To guarantee the safety, if proof cannot be obtained, we leave the checks untouched.

We highlight that while using external static analysis tools will add their running times as overhead to the compilation, very often, for safety critical programs (e.g., web servers), such cost may be justified by the better performance and security at runtime.

Finally, to address *Issue 3*, we build a general method of propagating information and assertions from different static analysis tools into the LLVM optimization chains.

III. DESIGN

To leverage the analysis power of current tools for MESCs, we design a methodology that connects the analysis tools with the safety instrumentation frameworks. This path is responsible for transporting analysis information related to runtime checks from external analysis tools, through a compiler's front and back-end, to the code that implements the safety instrumentation (Figure 1). We highlight that while some information-propagation infrastructure exists inside compilers to transport information from the front-end to the backend (e.g., # pragma directives or profiling metadata), this is highly specialized and can be used only for very specific optimization purposes. Our framework, in contrast, provides a general way to bring *any* analysis information generated by external analyzers, to *any* safety instrumentation implementation, where such information can be used.

In a larger context, not explored in this paper, our framework may be used to open a path among external analyzers and other back-end optimizations, further enhancing them by providing high quality analysis information, which is not available to a production compiler.

Requirements and issues. Our design is required to preserve the safety of the checks inserted by the MESCs. This must be guaranteed by the established soundness of the external tool analysis, restricting the choice of these tools. Given sound analysis, we must then bridge the semantic gap between tool output, and the assertion descriptions to be injected into the code: each tool has its own representation for analysis results, and these must be translated to a form usable by the compiler. Finally, the representation within the compiler of these analyses must not interfere with the ordinary work of the compiler.

```
1  int* p = (int*) malloc(100*sizeof(int));
2  int i=0;
3  while(i<100) {
4    p[i] = i*i;
5    i++;
6  }
7  //....
8  free(p);
9  //....
10 for(i=0;i<100;i++)
11   p[i] = 0;
```

Listing 1: The running example in C source

```
1  if(shadow(i)!=0) throw_UAF_Exception();
2  while(i<100) {
3    if(i< 0 || i >= getSize(p))
4          throw_OOB_Exception();
5    if(shadow(p[i]) !=0) throw_UAF_Exception();
6    t = multiply.with.overflow(i, i);
7    if(t.overflow == true)
8      throw_IOF_Exception();
9    p[i] = t;
10   i++;
11 }
```

Listing 2: Program snippet from the running example (the `for` loop is not shown) with runtime checks

Overview. A high level view of the main components and steps along this path is depicted in Figure 1. In the first step, C programs are given as input to external analyzers, which produce facts and information useful for removing unnecessary runtime checks; this information is encoded as *assertions*. For instance, if a runtime check's purpose is to catch out of bounds access, the information in the assertions is about object boundaries and variable ranges. In the next step, the assertions are given as input to the compiler's front-end and are transported through the parsing and Intermediate representation (IR) code generation phases to the back-end. In the back-end, the information contained in the assertions is associated with the intermediate code and transported through the chain of optimizations to the instrumentation framework, where it is finally used to remove checks.

After introducing a running example, in the rest of this section, we provide details about each of these steps.

Running Example. To illustrate our approach, we provide a simple running example in Listing 1 containing several operations that are instrumented with runtime checks. In reality, these runtime checks are inserted in the intermediate code generated by the front end, however, we show them in the source code for clarity. In Line 4, the variable `i` is used as an index into the array starting at `p`, where such access might cause an out of bounds access for values of `i` greater than 100. In the same line, `i` is also used as the operands of a multiplication, whose result may cause integer overflow for large values of `i`. Finally, in Line 8, the variable p is deallocated, and then used in Line 11. To prevent memory safety errors, MESCs insert runtime checks into the program. Listing 2 illustrates such checks, related to the `while` loop from Listing 1.

Lines 1 and 5 of Listing 2 contain checks inserted by

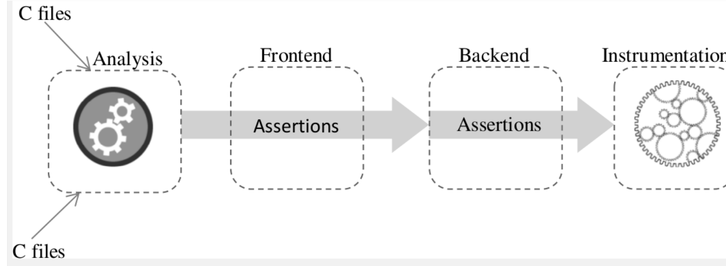

Fig. 1: Steps on the Pathway from the Analysis Tools to the Safety Instrumentation

Address Sanitizer. Address Sanitizer creates for every memory region, such as the memory region where `p[i]` is allocated, a shadow memory location, which contains the status of the program's region. When the program's memory region is not valid anymore (e.g., because of `free`), Address Sanitizer sets the shadow region to a non-zero value. In our example, the check in Line 5 computes the shadow memory location associated with the variable `p[i]`, which is used inside the condition of the `while` loop, and checks that it is still allocated.

Lines 3-4 contain another runtime check, inserted by Safecode. The check verifies that the access is within the bounds of the array pointed by `p` and throws an exception if this is not true. The function `getSize` exemplifies the Safecode operations to retrieve the runtime size of the object pointed to by `p`. Finally, Lines 6-8 contain the code transformed by the Integer Overflow Check instrumentation. Here, the original multiplication is substituted with a safe multiplication function, which returns a structure containing the result and a flag indicating if overflow occurred.

Evidently, the runtime checks inserted in Listing 2 by the safety instrumentations are not necessary. In fact, the value of `i` is between 0 and 100 for every possible run, therefore out of bounds checks and integer overflow checks are not necessary. In addition, the use of `p[i]` inside the `while` loop occurs before the `free`, therefore a check for detecting use after free is not necessary for that use.

We highlight at this point that existing compiler optimizations, such as `O3`, are, in general, not able to determine if the runtime checks are unnecessary. This is due to several reasons. First, many runtime checks are implemented as calls to functions that are not available at compile time but are linked with the code in a later phase. In addition, due to efficiency constraints, the algorithms used by these optimizers to analyze the code do not perform an analysis as deep as other static analysis tools who do not have those constraints. Additional optimizations performed by MESCs, are in general concerned with other aspects of the implementation. For instance, Safecode uses several link time optimizations to remove checks on bounds checks to single-element objects (e.g., scalars, or single element arrays), or uses caching of previously accessed arrays in the look-ups, so that a bounds check that has already been performed is removed.

Our framework brings such information from static analysis tools to MESCs, improving the performance of target programs by reducing unnecessary runtime checks. In the following subsections, we present different steps in this path: carrying assertions from static analysis tools to the compiler back-end, and leveraging the compiler to use the provided assertions to produce optimized code.

A. Deep Analysis

The goal of this step is to use external analyzers to produce information about a program, which can be used to remove unnecessary runtime checks. To remove runtime checks dedicated to preventing *out of bounds* memory accesses and *use after free* bugs, we use two external analysis tools: Frama-C [12], [13] and CodeSurfer [14], [15]. These tools exhibit several advantages with respect to LLVM's analysis capabilities. They can perform whole program analysis, spanning multiple compilation units and procedures. Furthermore, they are not as constrained as LLVM with respect to performance, and can perform a deeper and more complex analysis.

Frama-C. This is an analysis framework for C programs, which can be extended by many plugins that perform different types of analysis [5]. One of its most widely used plugins is the *Value Analysis* plugin, which derives the value ranges that variables can assume at runtime using abstract interpretation. When available, these ranges can determine at compile time if an out of bounds access is possible. The results of Frama-C's value analysis are guaranteed to be sound [13]. For instance, in Listing 1, Frama-C is able to determine that the range of the index `i` is between 0 and 100 for every possible program execution. Therefore, the access to the array in Line 4 will never be out of bounds and the out of bounds check for that operation can be removed. Furthermore, using the same bounds information derived by Frama-C, we can infer that the result of the multiplication in Line 4 can never overflow and therefore an integer overflow check related to that operation can be removed.

CodeSurfer. This framework provides different types of facilities for analyzing full programs, as well as an API to build plugins for customized queries over the code. After the code is parsed, several program representations are built, including full program abstract syntax trees, control flow graphs, and system dependency graphs. The latter represent the data and

[5]http://frama-c.com/plugins.html

control dependencies among program points. If the value of a variable at a point A depends on operations carried out at a point B, there is a *data dependency edge* from B to A. If execution of A depends on some condition at a point C, then there is a *control dependency edge* from C to A.

Data and control dependencies provide valuable information about the possible order of execution of program points. A data dependency edge between points A and B means there is at least one path in the control flow graph where A is executed before B. This precedence information establishes a relative order of execution between program points containing `free` statements and statements where pointers are used, which is used to determine if the use of a variable appears before or after a `free` statement.

For instance, in Listing 1, executing a backward slice query from the program point at Line 8, returns the program points for Lines 1 and 4, which contain the variable `p` used in Line 8. A forward slice query instead returns the program point 11, which is affected by the execution of Line 8. Once these dependencies are discovered, they can be used to assert that any use of a pointer that does not have a dependency from a `free` statement is safe; therefore the runtime check associated with that use can be removed.

To use these programs as external analyzers in our framework, there are several issues that need to be resolved.

Information granularity. One of the main problems in using external analyzers for removing checks is that of the granularity between the operations and information produced by these analyzers, and the operations of the safety instrumentation frameworks. In fact, the latter usually operate at the LLVM IR level, while the former at the source code level. For instance, Safecode instruments with runtime checks the LLVM IR code, while the Frama-C analyzer performs abstract interpretation on the C source code and computes the variables' value ranges.

To deal with this problem, we fix the granularity to the source variables' *uses* and *definitions*, and thus retrieve from the analyzers information about variable uses and definitions. The advantage of this choice is that these uses are readily available to the instrumentation framework of the compiler.

Analysis soundness. External static analyzers, in general, employ several approximations leading to false positives and negatives. To ensure that our framework does not violate the safety provided by the instrumentation frameworks, only sound analysis results are used to remove checks. The practical effect of this choice is that a significant percentage of runtime checks may not be removed.

B. Annotation Capture

Once the information about variable uses and definitions is produced by the external analyzers, it needs to be propagated to the back-end. There are several challenges to address in this step.

Language Heterogeneity. The information derived from external analysis must track the compiler mapping from source code language to the intermediate representation. Due to the static single assignment (SSA) nature of the LLVM intermediate representation (IR) language, one source code variable can be mapped to multiple IR variables, and a source statement may be split into different statements. We solve this problem by using debug information, which contains a mapping between source and LLVM variables, as well as by injecting the assertions as special constant strings in the program (see Listing 4).

Analysis Format. Another issue in this step is integration of the analysis results from different external analyzers in a common representation format, which can be used for different instrumentations. In particular, we design two types of assertions, one expressing the range of every variable use, and another expressing the safety of that use with respect to `free` statements. We provide more details about these assertions in Section IV-B.

C. Transport to the Backend

Once the assertions are available to the back-end, they are propagated through the chain of optimizations to the instrumentation passes. The main challenge in this step is the fact that the optimizations along the chain may change the code by transforming and removing instructions. For example, consider LLVM's `mem2reg` optimization, which minimizes the traffic between memory and registers by removing unnecessary `load` and `store` statements, by inserting `phi` nodes into the code, and so on. This code transformation modifies the mappings between source and LLVM variables discovered in the previous step, thus invalidating the assertions.

A general solution to this problem is provided in a prior work [16]. In general, to ensure mapping correctness, witnesses to the transformations performed by the optimization passes can be inserted for each pass with very little implementation overhead. These witnesses are responsible for relating the target and the source program after every optimization and can be used to update the assertions correspondingly. However, in this paper, we do not use such solution, since the code is not transformed before it reaches the instrumentation passes, and thus the assertions are always correct.

Two of the more problematic transformations for our approach are the introduction of temporary variables in LLVM to hold intermediate results or the deletion of LLVM instructions. To deal with this problem, we try to associate as many instructions as possible with the corresponding assertions, so that even when instructions are deleted we can recover assertion information from the remaining instructions. In addition, we compute new assertions by using existing ones. For instance, if the range of two variables is known, the range of their multiplication is computed and an assertion about this range is attached to the LLVM variable that stores the result of the multiplication. We provide more details about this task in Section IV-C.

D. Check Elimination

When the assertions reach the instrumentation passes, the information they contain determines whether a check is needed.

In particular, to remove Safecode `out of bounds` checks, we use two types of information: 1) the range information associated with the variables and discovered by Frama-C, and 2) the size of arrays (when known at compile time). If such information is known, then the check is as simple as determining if the values contained in range of the variable fall within the array size. When this check is positive, we can avoid the insertion of the runtime check.

For Address Sanitizer, a similar elimination procedure is used. In this case, the information about the safety of a pointer use is already available and no further computations need to be done. If a memory access is associated with an assertion that claims that it is safe, we skip the runtime check insertion. For instance, Listing 3 shows the code resulting from removing the unnecessary checks. As can be seen, the only remaining check is that in Lines 9-10, since there is a dependency between Line 6, where the variable `p` is freed and Line 11, where it is used.

```
1  while(i<100) {
2    p[i] = i*i;
3    i++;
4  }
5  //...
6  free(p);
7  //....
8  for(i=0;i<100;i++){
9    if(shadow(p[i]) !=0)
10     throw_UAF_Exception();
11   p[i] = 0; //use after free
12 }
```

Listing 3: Optimized program after leveraging assertions

IV. Implementation

Our implementation is built on top of LLVM 3.4 (We used version 3.2 for Safecode due to compatibility). LLVM is a popular compiler back-end, which is used to perform a wide range of optimizations and final code generation. The optimizations in LLVM are structured as a sequence of passes that operate on an intermediate representation of the code called LLVM bytecode (LLVM IR), which is generated by the compiler's front-end. The safety instrumentations are built as additional passes on top of the other passes in LLVM. They operate by intercepting LLVM `load` and `store` statements (Safecode and Address Sanitizer), which load and store values from memory locations to registers, and by inserting runtime checks before those statements. Our implementation is composed of a series of tools that extend this architecture. A high level overview of our architecture is shown in Figure 2.

In the first step of the implementation architecture, the program files are given as input to Frama-C or CodeSurfer. In particular, the Value Analysis Plugin of Frama-C is used to derive the variable ranges and remove the bounds and integer overflow checks, while CodeSurfer to remove the use after free checks. These two tools have been extended by plugins that we wrote to perform the analysis and to generate analysis results useful for removing unnecessary checks. The results are output from each tool separately as lists of assertions associated with a line number and a file name. Next, our rewriter uses this list of assertions to inject them in the source code at the specified line number. The output is an annotated file, as shown in Listing 4 where each of the lines 2-3, 6-8, 10, and 17, contains an assertion about the use of the variables in the first line of the original code following those assertions.

Next, the annotated C source file is passed as input to Clang, the LLVM compiler front-end, which translates it to LLVM IR, without performing any code optimizations. The resulting LLVM IR is next passed in input to an LLVM pass that we develop, the *Assertion Mapper* pass, which is responsible for mapping the source variables to the LLVM variables and associating the source assertions with the LLVM instructions. The output of this step is an annotated LLVM IR file, where the annotations contain the assertions about the value ranges and the uses before the free statements. Finally, the LLVM bytecode is given as input to the safety instrumentation passes, Safecode and ASAN (Address Sanitizer), which have been modified to read the annotations and use them to avoid the insertion of unnecessary runtime checks.

We next provide details about each of these steps.

A. Analyzers Implementation

Frama-C plugin Implementation. Frama-C framework [12], [13] analyzes C source files and computes the value-range information of variables. The computation is performed by a Frama-C built-in plugin, the Value Analysis plugin. However, the computed ranges are internal to the Frama-C framework and cannot be explicitly extracted. To perform such extraction, we implemented a Frama-C plugin using the APIs provided by this tool. This plugin visits every instruction in the AST tree inside Frama-C, extracts all the variables at each node, and queries the value analysis plugin for each variable to get the corresponding value ranges. The results are then stored in an annotation specification file to be used later by the rewriter. The specification and the rewriter are described in subsection IV-B below.

CodeSurfer plugin Implementation. To ultimately remove the checks inserted by Address Sanitizer we built a plugin for CodeSurfer that executes the following two tasks:

1) Using the data dependency graph, we issue backward and forward slicing queries to find the order of execution among statements containing a call to a `free` and statements that use the pointer being freed. If there exists a dependency edge from a statement `free(p)` to a statement where pointer `p` (or any of its aliases) is used, then there exists a potential use after free vulnerability. This implies that the Address Sanitizer's checks inserted at those uses must not be removed. If, on the other hand, there exists a dependency between a statement where a pointer `p` (or any of its aliases) is used and a statement `free(p)`, then the statement is executed before the `free(p)` along some path. Finally, if dependencies exist in both directions, then the two statements can be executed in any order.
2) Find the program points that contain uses that are not related to a `free(p)` statement. These include all those

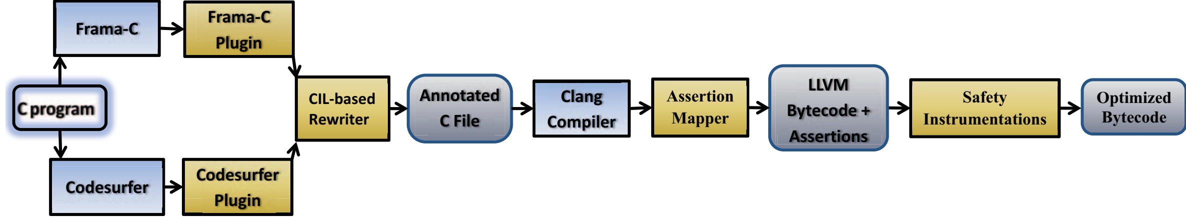

Fig. 2: Implementation Architecture.

uses for which runtime checks are inserted by Address Sanitizer, but which are not usually freed in a program (e.g., non pointer variable uses).

The plugin uses Codesurfer's APIs to execute both tasks and outputs a file of annotations about the safety of the program points, together with line numbers. The output of the plugin is an assertion file associated with every source code file. Each assertion contains the safety information of the corresponding variable use, together with the line number where that use occurs.

B. *From the Analyzers to the Frontend*

To provide a common framework for both the `out of bounds` and `use after free` optimizations, we specify an assertion language to express value-range and memory safety information. This specification is designed to represent value-range and memory safety assertions about variables in each program location. The syntax of our specification is described in Table II. The basis syntax includes file name, line of code, and assertions. We need both file name and line of code to instrument the assertions in the right place. There are two type of assertions: (1) value-range assertions represents the value range and (2) safety memory of a specific variable.

<assertion_spec>	::=	filename:lineofcode#assertions
<assertions>	::=	(@assert <assertion>;)+
<assertion>	::=	<value_assertion>
		| <safety_assertion>
<value_assertion>	::=	<expression> ('&&'<expression>)*
		| <expression> ('||'<expression>)*
<expression>	::=	<variable> op <value>
op	::=	$'=='$ | $'>='$ | $'<='$
<safety_assertion>	::=	$safe$(<variable>)' =' boolean
<variable>	::=	<string_literal>
<filename>	::=	<string_literal>
<lineofcode>	::=	integer
<value>	::=	integer | real

TABLE II: Syntax of the common assertion language

As our framework is designed to use different analysis tools, we need a consistent way to introduce assertions into code. To this end, we develop a transformation tool to embed assertions into the program in the form of string variables, which are specially named to avoid interference with the existing program variables. These variables are injected before the corresponding instructions in the source, and are propagated to the compiler back-end through the standard code generation phase of the Clang front-end.

```
1  int* p = (int*) malloc(100*sizeof(int));
2  char* assert1= ``@assert i==0'';
3  char* assert2= ``@assert safe(i) = true'';
4  int i=0;
5  while(i<100) {
6    char* assert3= ``@assert i>=0 && i <100'';
7    char* assert4= ``@assert safe(p) = true'';
8    char* assert5= ``@assert safe(i) = true'';
9    p[i] = i*i;
10   char* assert6= ``@assert i>=0 && i <100'';
11   i++;
12 }
13 ....
14 free(p);
15 ....
16 for(i=0;i<100;i++){
17   char* assert7= ``@assert safe(p)= false'';
18   p[i] = 0; //use after free
19 }
```

Listing 4: Program from the running example with injected assertions

This transformation is based on CIL (C Intermediate Language) [17]. CIL is a high-level representation of C programs that is lower-level than AST and higher-level than typical intermediate languages. CIL has a set of tools for static analysis and transformation of a valid C program using a few core constructs with clean semantics. Our CIL plugin takes as input a C source file and the corresponding set of assertion specifications, and visits all instructions in that C source file and injects the assertions. Listing 4 demonstrates the output of our transformation tool that inject assertions in string variables in corresponding source location.

C. *Backend Implementation*

Since the assertions are inserted in the source code file as assignments to string variables, these assignments are translated by the Clang frontend together with the rest of the program. To attach the assertions to the LLVM IR code, so that they are available to the instrumentation passes, we designed an *Assertion Mapper* LLVM pass. This pass is run immediately after the code generation phase of Clang, before any other optimization passes. In fact, since the assertion assignments are semantically orthogonal to the rest of the program and not used anywhere else, they may be removed by

```
1  %1 = load %i
2  %2 = load %i
3  %3 = mul %1, %2
4  %4 = load %i
5  %5 = load %p
6  %6 = GEP(%5, %4)
7  store %3, %6
```
Listing 5: LLVM IR code compiled from the running example

the optimization passes as dead code. The *Assertion Mapper* pass works according to the steps described below.

Source-IR mapping. The *Assertion Mapper*'s first job is the creation of a mapping between all source code variables and the corresponding LLVM allocated memory locations. This mapping is necessary for associating assertions with the correct instructions in the LLVM IR code. This mapping is created by using the debug information contained in the code, which provides the name of the source variable for every LLVM allocated memory location.

For instance, in Listing 5, we show the portion of the LLVM IR code corresponding to the assignment `p[i] = i*i;` in the source code. In the Listing, identifiers start with a % sign. In Lines 1-2, there are two copies of the variable `%i` loaded into two registers `%1` and `%2` before the multiplication in Line 3. Next, the value of the pointer `p` is loaded into a register `%5`, and the pointer to the i-th element starting from `p` is obtained through the `GetElementPointer` (GEP) instruction. Finally, the result of the multiplication is stored into that element.

As can be noted from the example, there are several copies of the value of the variable `i`, each one having a different name. Therefore, an assertion about `i` (e.g., an assertion that the range of `i` is between 0 and 100) is valid for all those copies and needs to be associated with all of them. To solve this issue, we use the debug information, which contains a mapping among C source variables and LLVM IR memory locations.

Metadata attachment. Next, for every `load`, `store` and `gep` instruction, the corresponding assertions are extracted from the code, with the help of the previously created mapping, and attached to those instructions as LLVM metadata. An example of the output of this step is shown in Listing 6. For every LLVM instruction, the text after the `!` shows the safecode-related metadata. These contain the range information corresponding to the value contained in the LLVM identifier. For instance, the metadata associated to the variable `%4` in line 4 provides the range of the array index, and the metadata associated to line 5 provides the size of the array. The `GEP` instruction returns a pointer to the element indexed by the variable `%4` inside the array starting at the memory address pointed by `%5`.

The metadata related to the Address Sanitizer's check are similarly attached to `load` and `store` statements. These

```
1  %1 = load %i !``%1 >= 0 && %1 < 100''
2  %2 = load %i !``%2 >= 0 && %2 < 100''
3  %3 = mul %1, %2 !``%3 >= 0 && %3 < 10000''
4  %4 = load %i !``%4 >= 0 && %4 < 100''
5  %5 = load %p !``size(%5) = 100''
6  %6 = GEP(%5, %4)
7  store %3, %6
```
Listing 6: LLVM IR Annotated with Assertion Metadata

metadata simply contain information about the safety of the instruction.

Metadata propagation. An additional task of the *Assertion Mapper* is to propagate metadata to the temporary variables that appear in the LLVM IR. In particular, given an instruction, *Assertion Mapper* checks if the operands of the instruction contain any metadata, and if possible, merges those metadata using the same semantics of the instruction and assigns the new metadata to the result of the operation. For instance, in Listing 6, the assertions about the variables `%1` and `%2`, are used to derive the assertion attached to the multiplication result in Line 3. Currently, *Assertion Mapper* supports this propagation for LLVM's arithmetic and sign extension operations.

D. Check Removal Implementation

To remove the insertion of run time checks by Safecode and Address Sanitizer, we modify the code of the corresponding LLVM passes. Our modification includes additional code that intercepts the same `load`, `store`, and `gep` instructions intercepted by these passes and reads the metadata associated with those instructions.

Safecode checks removal. Safecode uses several LLVM passes for adding the `out of bounds` run time checks. These passes are responsible for tracking the allocated regions, storing their sizes, and checking that every access to those regions does not fall outside of the bounds. In particular, for every array access, the run time check inserted by Safecode ensures that the pointer to the referenced array element does not fall out of the array bounds.

In our implementation, we modify each of these passes. In particular, our implementation starts by reading the range associated with an array index from the metadata and retrieves the size of the array using the LLVM API. For instance, in Listing 6, the metadata associated with the arguments (`%4`, `%5`) of the LLVM `GEP` instruction, contain the size of the array and the range of the index. For fixed sized arrays that are allocated inside the same function as the array access (either on the stack or on the heap), the size information is readily available. For arrays that are passed as parameters in input to a function, the size determination is more complex. In fact, the array may have been allocated in any of the callers of that function or any of its predecessors in the function call graph, and it may have been passed in as a parameter along the sequence of function calls. That is, at run time the array may have any number of sizes depending on the caller. To

retrieve the array size, we travel backwards one step in the function call graph and retrieve the possible array sizes. If the sizes can be retrieved this way, we use the minimum size, as the safest option. When the size of the array cannot be determined in this way, we choose the safest course of action and do not remove the run time check.

Integer overflow check removal. The implementation of the integer overflow checks removal is very similar to that of Safecode. In particular, our implementation removes the checks that are inserted by the `-fsanitize=signed-integer-overflow` and `-fsanitize=unsigned-integer-overflow` options of Clang. These checks are inserted by the frontend, which, based on the signedness of the operands, replaces several arithmetic operations with equivalent LLVM intrinsics during the code generation. For instance, additions of signed integers are replaced with `llvm.sadd.with.overflow`. In addition, the pass inserts a check over the overflow flag set by the LLVM intrinsic. In our implementation, we intercept every such intrinsic accompanied by a check of its overflow flag, and, if the variable ranges of the operands are available, we perform the same arithmetic operation using the value ranges as operands. Next, we compare the resulting value range with the maximum integer of the framework and remove the check if all the values of the range fall below that maximum integer. In some cases, Frama-C's value analysis produces ranges that include $-\infty$ or $+\infty$ or both. In these cases, we do not remove the integer overflow checks.

Use after free check removal. The implementation of this check is fairly simple. In the same way as for the other instrumentations, we intercept every `load` and `store` instruction that Address Sanitizer intercepts. Next, we read the metadata information that tells us if the instruction is safe. In this case, we skip the check insertion.

V. Evaluation

A. Setup

The results of our approach are shown below. Our framework incorporates out-of-bounds runtime checks inserted by LLVM passes for Safecode, address sanitizer, and signed and unsigned integer overflow sanitizer. The evaluation was done by instrumenting each benchmark program to track and output user time for the course of its execution. Each program was run at least ten times and the times were averaged. Our runtime tests were performed on a GNU-Linux machine running the LINUX Ubuntu distribution 12.04, on an Intel Xeon CPU at 2.40GHz.

B. Benchmarks

We selected open source test applications that cover a range of sizes and operational characteristics. Some applications were CPU-intensive, like the matrix manipulation and equation solver programs, and some were I/O intensive, like the media conversion and file compression utilities. Half the test programs had small line counts, with less than 2000 non-commentary source lines (CLOC); half were larger applications with thousands of non-commentary source lines. To illustrate the optimizations of our framework, we selected benchmarks with a wide range of non-commentary source lines, also looking for many (hundreds) of array references.

Benchmark	Description	Line Count	Automatic Annotation	% Lines Annotated
oggenc	Audio Compression Utility	48347	880	1.82%
Laspack	Solve large sparse systems of equations	7656	100	1.31%
gzip	File compression utility	5352	1451	27.11%
debie1	Analysis of Micro-Meteoroid Impacts	5243	1279	24.39%
bzip2	Block-Sorting File Compressor	5115	563	11.01%
appbt	Differential Equation Solver	3047	10	0.33%
susan	Image processing	1463	109	7.45%
quicklz	Fast File Compression Utility	870	64	7.36%
cpumax	Simple Add Instructions	585	15	2.56%
linpack	Measure system floating point computing power	579	166	28.67%
NEC-Matrix	Matrix operation with a fixed size	113	70	61.95%

TABLE III: Benchmark Source Size and Annotation Coverage

Table III illustrates the source code line counts and the percentage of source lines that were annotated by Frama-C, with variable value range information. The chart is ordered from largest to smallest total CLOC; the display is divided between two groups of large and small line counts, in this table and in the next several figures.

There is a weak anti-correlation between line counts and annotation coverage. Typically, no more than 25% of the source lines were annotated; for some applications the percentage is very low, 1-2%. The smallest benchmark, NEC-matrix, achieved the highest annotation rate, at 62%. In the discussion to follow we find it useful to divide the benchmarks into large and small groups, with a cutoff at around 2000 CLOC. Generally there are higher percentages of annotated lines in the smaller benchmarks, with some exceptions. The two largest benchmarks have among the lowest annotation rates, due to large portions of the programs depending on runtime values not available to Frama-C. In such cases the abstract interpretation finds relatively few instances of sound results. This in turn impacts the number of checks that are removed, since we only remove checks related to annotations.

C. Annotation Analysis

The next three figures illustrate strategically removing runtime checks that our analysis proved were safe to remove, thereby improving benchmark performance. Each chart shows benchmarks ordered from the largest to the smallest. Two metrics are given in the charts: the percentage of runtime

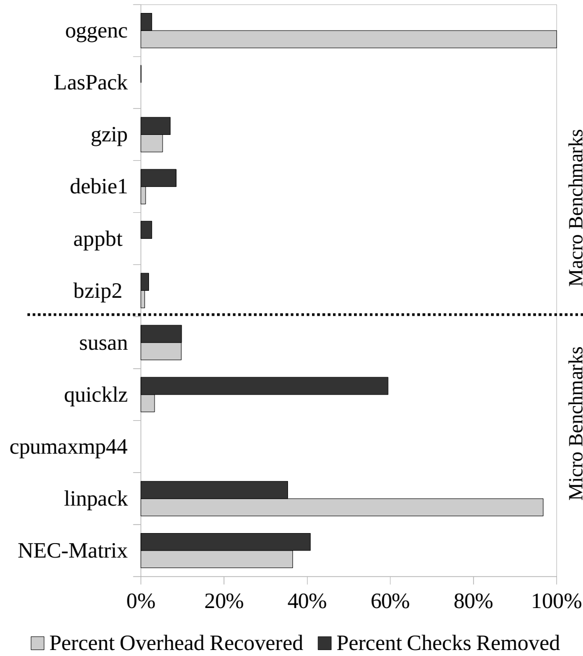

Fig. 3: Safecode Benchmarks Performance

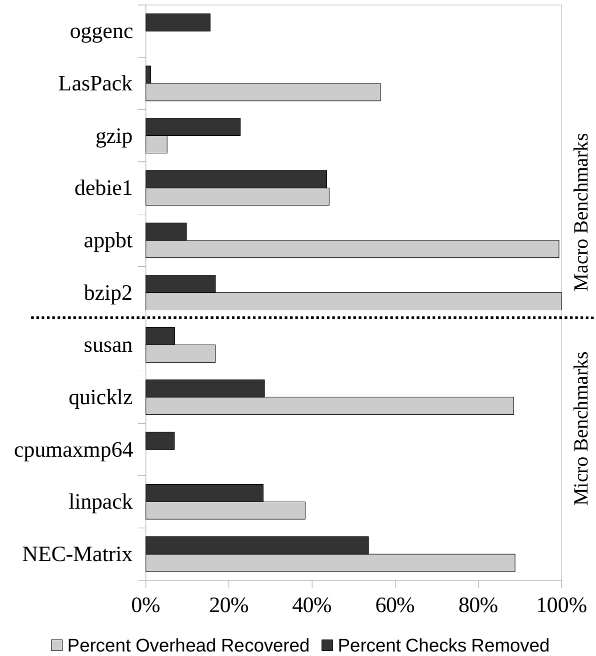

Fig. 4: Integer Overflow Check Benchmark Performance

checks that were *removed* by our protocol, and the percentage of the performance overhead due to runtime checks that was recovered by the framework:

$$\%\text{Overhead_Removed} = 1 - \frac{R - O}{C - O}$$

where R is the performance with checks removed, C is the performance with all checks present, and O is the performance with no checking.

Safecode. Figure 3 shows results following removal of Safecode checks. We note that those checks, added by Safecode, are not removed by the `O0-3` optimizations. This class of runtime checks had the most severe effect on performance overhead, as seen above in Table I. Measuring the performance benefits of check removal, we see mixed results, ranging from no recovery of overhead, all the way to 100% recovery. The latter is associated with the benchmark with the smallest measured overhead, so its impact on absolute program effort is not as great as it is for programs with larger overhead penalties. We attribute results showing no overhead recovery to the removal of safety checks in regions of code that are seldom executed, such as initialization code. Results with significant overhead recovery are attributed to checks removal in frequently executed code, particularly program loops. This effect was manually verified in some smaller benchmarks. There is not close tracking between the percentage of checks removed and the percentage of runtime recovery, since there is wide variation over whether removed checks are in frequently executed sections of code.

Integer Overflow. In the experiments on integer overflow (Figure 4), the percentage of checks removed and the performance improvement due to removal of checks are more substantial than was seen in the Safecode experiments; again, the values are quite variable across all the benchmarks. Here the improvements in performance roughly correlate with removal of runtime checks, with some exceptions. We note that these experiments were conducted using `O3`, so the improvements are after `O3` optimizations. The LasPack results illustrate how removal of a few percentage points of checks can recover most of the overhead; this strong benefit would be difficult to achieve by manual analysis.

For removal of integer overflow checks, not only are the overhead recovery measurements quite robust, with 7 of the 11 programs exceeding 40% removal, but overhead levels are much smaller than with Safecode. Therefore these results represent strong measurable gains in secure program performance.

Address Sanitization. We also performed measurements on removal of checks inserted by the `sanitize=address` compiler pass. These checks are introduced for trapping use-after-free events; data is shown in Figure 5. As can be noted, the improvements in the run time overhead of Address Sanitizer range between 15-40%. Many of the address sanitization checks that are removed are associated with non-pointer variables. We note that a significant portion of Address Sanitizer's overhead depends also on another instrumentation of the program aimed at detecting out of bounds checks, which are not removed in the current implementation. Again, most benchmarks exhibit substantial recovery of overhead.

Summary. Results of our benchmark experiments are encouraging, although varying widely with the programs and sanitizations being tested. Our best runs show recovery of

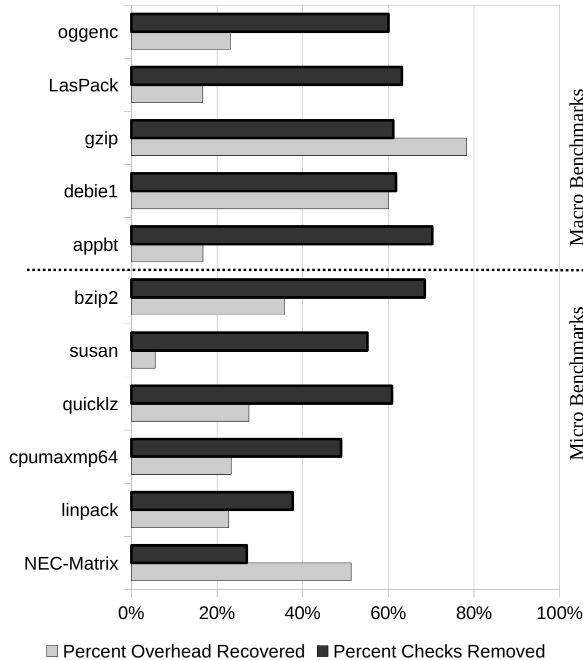

Fig. 5: Address Check Benchmark Performance

Benchmark	Safecode	IOC	ASAN
oggenc	0.28 ➔ 0.00	0.21 ➔ 0.21	3.48 ➔ 2.68
LasPack	30.30 ➔ 30.30	0.97 ➔ 0.42	4.29 ➔ 3.57
gzip	15.70 ➔ 14.88	0.22 ➔ 0.21	0.94 ➔ 0.20
debie1	46.12 ➔ 45.61	0.56 ➔ 0.31	4.46 ➔ 1.78
appbt	97.97 ➔ 98.20	4.85 ➔ 0.03	2.60 ➔ 2.17
bzip2	70.15 ➔ 69.52	0.39 ➔ 0.00	3.87 ➔ 2.49
susan	18.23 ➔ 16.46	2.35 ➔ 1.96	4.12➔ 3.89
quicklz	19.04 ➔ 18.41	0.59 ➔ 0.07	1.80 ➔ 1.30
cpumaxmp64	4.00 ➔ 4.00	0.09 ➔ 0.09	0.07 ➔ 0.05
linpack	28.00 ➔ 0.91	0.34 ➔ 0.21	3.44 ➔ 2.65
NEC-Matrix	55.67 ➔ 35.33	1.88 ➔ 0.21	4.63 ➔ 2.25

TABLE IV: Net Benchmark Overhead Following Check Elimination

more than half the overhead due to runtime security checks, while in a few cases there is no improvement. The net overhead for all the experiments is seen in Table IV, where results in each case are presented as `X ➔ Y`, where `X` is the overhead factor with all checks in place, and `Y` is the overhead factor after some checks were removed by our protocols. This presentation is more revealing than the charts in Figures 3, 4, and 5, which only illustrate the relative improvements due to checks removal. Table IV gives the absolute overheads running the optimized programs, compared with the performance of the original, unsafe code.

Table IV does not include the time spent by the static analyzers for deriving the assertions, since this is an operation that is carried out only once for every program. This time can also vary widely depending on the analysis depth of the static analysis. For instance, the `-slevel` option of Frama-C's value analysis specifies a limit on the amount of loop unrolling, which can be varied depending on the required precision.

VI. Related Work

Memory safety is a widely studied problem and there exists a large body of work that addresses it. The large majority of this work proposes different schemes that instrument programs to detect and prevent memory errors at runtime [7], [8], [18], [10], [19], [20], [21], [22], [23], [24]. In these techniques, a runtime infrastructure is added on top of the programs to create, update, and query information about every memory access. These approaches deal both with "spatial memory safety", which prevents out of bounds memory errors such as buffer overflows, and "temporal memory safety", which prevents other memory errors dependent on order of execution, such as use-after-free and double-free.

These techniques can incur high overheads. This issue is widely recognized and several optimization efforts have been carried out. Almost all of these optimizations, however, deal with the efficiency of the runtime infrastructure added to the program. Address Sanitizer ([8]), for example, uses shadow memory, which computes the location of the status information very quickly; other approaches incorporate different efficient data structures ([25]). A recent approach in the direction of removing runtime checks is ASAP, which, given a budget on the maximal desired overhead, profiles the programs, ranks the runtime checks in order of their execution counts, and removes the most frequent ones [26]. However, this system makes no safety guarantees, and it may remove checks which are necessary for safety.

[16], [6] tackle the problem of propagating assertions in a compiler. [16] develops the theory using refinement relations and [6] provides a simple concrete instance of this approach. Our work is more comprehensive in this regard, by developing a detailed system design and implementation, applying it to several bounds checks and evaluating with large benchmarks.

Several verification efforts show that if the input code is free of memory errors, so is the output code [1], [2], [3], [4]. However, they do not sanitize errors in the input code.

Lee *et al.,* instrument the intermediate code to keep track of pointer aliases at runtime. In addition, the code is also instrumented to nullify all the aliases of a pointer when that pointer is freed [18].

Other techniques rely on changing the memory allocation layouts, so that memory safety errors do not occur, or occur with low likelihood [27], [21], [11]. Among these, Pool Allocation is a strategy to detect and prevent memory safety errors [11]. It relies on a type homogeneous allocation strategy (where variables of the same type are allocated in the same memory pool) to enable restrictions on the memory regions that are allocated and referenced. However, this strategy works only on a subset of the C language.

DieHard and Cling use additional memory space to decrease the likelihood of accessing previously allocated memory addresses [27], [21]. However, they come with a high memory usage overhead.

Additional tools, created in the context of program debugging, can be used to detect memory errors. Among these, Valgrind's Memcheck [28] and Electric Fence [29] also have a very high overhead both in memory and running time.

Similarly to our approach, `USHER` [30], analyses programs at compile time to remove unnecessary checks inserted by MemorySanitizer (http://clang.llvm.org/docs/MemorySanitizer.html) to detect uninitialized reads. However, they build their own analysis framework for this specific problem inside LLVM, while in our approach, we use existing tools to feed analysis information inside the LLVM backend.

VII. Conclusions

In this paper, we present a framework for improving the performance of programs instrumented with run time checks. Our framework uses external analysis tools to complement the compiler's analysis and provide information for proving spatial and temporal safety of memory operations. Our contribution is providing a mechanism to transmit constraint information discovered by the external tools through the compiler phases, to explicitly target removal of unnecessary runtime checks. This mechanism significantly alleviates much of the performance burden due to incorporation of memory safety checks, and is a significant step towards acceptance of compiler-based security defenses in production software.

Acknowledgments

This material is based on research sponsored by DARPA under agreement number FA8750-12-C-0166. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. Phu H. Phung has been partially sponsored by grants from University of Dayton Research Council and the Swedish Research Council (VR) through Chalmers University of Technology.

References

[1] N. Benton, "Simple relational correctness proofs for static analyses and program transformations," in *POPL*, 2004, pp. 14–25.

[2] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," in *POPL*. ACM, 2006, pp. 42–54.

[3] X. Leroy, "Formal Verification of a Realistic Compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009.

[4] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Formal verification of ssa-based optimizations for LLVM," in *ACM SIGPLAN PLDI '13*. ACM, pp. 175–186.

[5] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, pp. 477–526, May 2005.

[6] R. Gjomemo, K. S. Namjoshi, P. H. Phung, V. N. Venkatakrishnan, and L. D. Zuck, "From Verification to Optimizations," in *VMCAI 2015*, ser. Lecture Notes in Computer Science, vol. 8931. Springer, 2015, pp. 300–317.

[7] D. Dhurjati and V. Adve, "Backwards-Compatible Array Bounds Checking for C with Very Low Overhead," in *Proceedings of the 2006 International Conference on Software Engineering (ICSE'06)*, May 2006.

[8] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *2012 USENIX Conference on Annual Technical Conference (USENIX ATC 2012)*. USENIX Association, pp. 28–28.

[9] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in C/C++," ICSE '12. IEEE Press, 2012, pp. 760–770.

[10] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in PLDI 2009. ACM, pp. 245–258.

[11] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner, "Memory safety without runtime checks or garbage collection," in *2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems(LCTES 2003)*. ACM, pp. 69–80.

[12] "Frama-c," http://frama-c.com/, 2013.

[13] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," in *10th International Conference on Software Engineering and Formal Methods (SEFM 2012)*. Springer-Verlag, 2012, pp. 233–247.

[14] "CodeSurfer," http://www.grammatech.com/research/technologies/codesurfer.

[15] T. Teitelbaum, "Codesurfer," *ACM SIGSOFT Software Engineering Notes*, vol. 25, no. 1, 2000.

[16] K. S. Namjoshi and L. D. Zuck, "Witnessing program transformations," in *Proc. 20th Static Analysis Symposium*, ser. LNCS, vol. 7935, 2013, pp. 304–323.

[17] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *11th International Conference on Compiler Construction (CC)*. Springer-Verlag, 2002, pp. 213–228.

[18] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015*. The Internet Society, 2015.

[19] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Cets: Compiler enforced temporal safety for C," in *Proceedings of the 2010 International Symposium on Memory Management*, ser. ISMM '10. ACM, 2010, pp. 31–40.

[20] W. Xu, D. C. DuVarney, and R. Sekar, "An efficient and backwards-compatible transformation to ensure memory safety of C programs," in *12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering (SIGSOFT '04/FSE-12)*. ACM, pp. 117–126.

[21] P. Akritidis, "Cling: A memory allocator to mitigate dangling pointers," in *Proceedings of the 19th USENIX Conference on Security (USENIX Security 2010)*. USENIX Association, 2010.

[22] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *ISSTA 2012*. ACM, pp. 133–143.

[23] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *22nd USENIX Conference on Security*. USENIX Association, 2013, pp. 337–352.

[24] D. Bruening and Q. Zhao, "Practical memory checking with Dr. Memory," in *CGO 2011*. IEEE Computer Society, 2011, pp. 213–223.

[25] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *18th Conference on USENIX Security*. USENIX Association, 2009, pp. 51–66.

[26] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder, "High system-code security with low overhead," in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 866–879.

[27] E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," *SIGPLAN Not.*, vol. 41, no. 6, pp. 158–168, Jun. 2006.

[28] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *PLDI 2007*. ACM, pp. 89–100.

[29] N. Joukov, A. Kashyap, G. Sivathanu, and E. Zadok, "Kefence: An electric fence for kernel buffers," in *StorageSS 2005* . ACM, November 2005, pp. 37–43.

[30] D. Ye, Y. Sui, and J. Xue, "Accelerating dynamic detection of uses of undefined values with static value-flow analysis," in *CGO '14*. ACM, 2014, pp. 154–164.