Static Analysis To Improve Compiler Sanitization

Edmund W. Ballou

Masters Project

Advisor: V.N. Venkatakrishnan
Secondary Committee Member: Rigel Gjomemo
Graduation: December, 2015
UIN: 664068791

Abstract

Software security problems in inherently unsafe languages (C/C++) can be addressed by compiler tools that automatically insert checks that trap execution when exploitable faults are encountered at runtime. Such checks are seldom incorporated in production code because of the performance cost they impose, as the check injection to ensure safety is applied indiscriminately to every memory access. Our lab has developed static analysis methods to eliminate runtime checks that can be proven to be unnecessary. The benefit of this is to reduce overhead, improving security of running code at reasonable performance cost. In the current work we present results with benchmark programs that demonstrate the promise of this approach.

Using the static analysis frameworks frama-C and CodeSurfer, variables value constraints and code dependency traces are converted to annotations inserted in the source code. Customized LLVM compiler passes incorporate these annotations into variable-attached metadata. During passes that insert runtime security checks, this metadata validates skipping check insertion in instances where the variable value has been proven to be safe.

The LLVM compiler "safecode" pass which provides memory safety has so far proved only partially successful; improvements in this pass may manifest in the next iteration of our software. The LLVM compiler "integer overflow" pass exhibited excellent improvement in most of the benchmarks tested. In several benchmarks, over 80% of the runtime performance overhead due to these security checks was mitigated by our optimization protocol. Removal os "address sanitization" checks was also generally successful, delivering mitigation of more than 50% of the checks-imposed overhead for several benchmarks.

The safety checks we studied can effectively elimnate vulnerabilities due to buffer overflow, integer overflow, and use-after-free scenarios. Our protocol greatly enhances the usability of these compiler-generated runtime protections.

Contents

1 Introduction

Modern computer software requires both performance and security. Data processing burdens for applications are growing without bounds, as storage costs have plummeted. Although processing power has also expanded, the net performance demand continues to increase.

At the same time, society has become dependent on correct computer operations for many mission-critical purposes, while being plagued by software errors that are inherent and insidious, causing intrinsic failures and leaving openings for malicious interference. Security concerns about software vulnerabilities can be addressed by writing new code in or porting legacy code to hardened languages[16]. This approach shows promise for the future but does not help with massive amounts of current software investment.

Improving security in current and legacy software can be addressed by two approaches. Static analysis of the source code and various steps of compilation in the course of generating an executable file helps remove software errors, and provides analytic information about program behavior that reveals potential vulnerabilities. Dynamic code insertions perform runtime checks during program execution, and provide recovery opportunities when vulnerabilities targeted by the inserted checks are exposed.

Static analysis approaches are valuable for certain classes of error discovery as well as program optimization. Verification of the compiled code is under study by several labs [19, 18, 5, 32]. Although this approach hold promise, it is not scalable under state-of-the-art technologies.

Optimization is a principal function of modern compilers, which in the course of translating source code to executable files perform transformations to enhance performance.[1] However, analysis and transformation at compile time is constrained by performance during the development cycle, in which frequent, repeated compilations are required; and by the stringent need for correctness, which limits compiler manipulations to thoroughly tested, mature transformations.

Independently of compiler development, there has been evolution of stand-alone static analysis tools with a somewhat different orientation. These research tools provide deep program analysis mechanisms, are more advanced than compiler optimization processes, and offer opportunities for highly targeted transformations.

Dynamic security hardening seeks to remove software vulnerabilities, by automatically inserting extra code that protects a running program from being hijacked. For example, an attack that exploits the scenario in which integer overflow gives access to unexpected, unprotected memory regions is prevented by adding a check every time an arithmetic manipulation of an integer resulting in overflow is followed by an undefined operation; an inserted security check can then trap out of the module at risk. This can essentially eliminate this targeted class of vulnerabilities; however, it incurs substantial performance penalties, which can exceed 200%. Because these runtime checks to ensure secure operations are so resource-intensive, administrators, researchers and programmers are reluctant to apply these compiler options for software being deployed in the field.

As applications must scale to managing increasingly huge amounts of data, they are also subject to continuing attacks from malicious actors. Efforts to tighten security have resulted in coding improvements employing both static and dynamic approaches [27]. Although dynamic approaches have the advantage of providing real-time responses to unexpected and previously uncharacterized attacks, they incur performance penalties that are often too severe to be usable.

Investigations at the IGERT Laboratory at UIC-CS have experimented with capturing information from static analysis research tools. Using "value analysis" methods in frama-C, a framework for the analysis of C-language programs, we have successfully modified LLVM compiler code transformations to improve the performance of software that has been hardened by runtime checks. Source code annotations inserted by our custom "acslgen" plugin in frama-C are leveraged during compilation to substantially reduce the overhead of runtime security checks. This is accomplished by LLVM compiler modifications that capture the source code annotations during compilation and convert them to metadata attached to the pertinent variables. Subsequent passes that insert security checks are also modified so that variable range

[1] In some scenarios, source code that was written to enhance runtime security may be removed by optimizations[13].

information found in the metadata is applied to determine that particular variable operations are already memory safe, and so these checks are omitted. The current study extends the work of Gjomemo et al [14] and gives metrics on benchmark scenarios validating this approach. Portions of this report were jointly written with these authors: Rigel Gjomemo, Kedar S. Namjoshi, Phu H. Phung, V.N. Venkatakrishnan, and Lenore D. Zuck.

2 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 6.

Despite the intense level of focus, software vulnerabilities abound. One reason for this is the 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 second, related reason is performance concerns. 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. We call these compilers Memory Error Sanitization Compilers (MESCs). MESCs are an attractive solution to the critical mass adoption problem. MESCs have been developed for gcc (as of gcc 4.8), LLVM (from Clang 3.1) and Microsoft Visual Studio platforms. While these sanitization passes are rarely Incorporated in production software due to performance concerns, their functionality aids in testing, error detection and error diagnosis.

2.1 Analysis of runtime overheads

In 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 [14]. The three sanitization protocols, Safecode, Address, and Integer Overflow, are discussed in detail below. Similar conditions obtained as for the measurement runs reported in the Evaluation section. Our runtime data is shown in Table 1.

We find large overhead for most of the benchmarks, with some exhibiting a slowdown exceeding a factor of 50; a few showed modest performance cost, below 20%. These costs present a challenge to the security community, for runtime security checks to become acceptable in production software. The current work addresses this situation by designing strategies for targeted restriction of runtime checks insertion.

Benchmark	**Safecode**	**IOC**	**Address**
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	
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	1.88	4.63

Table 1: Benchmark Overhead due to Runtime Checks, for Three Sanitization Protocols

2.2 Optimizing runtime checks

Current efforts to improve performance of runtime checks in MESCs work with the runtime infrastructure through a variety of implementation strategies that involve the runtime data-structures. Examples include fat pointers [10, 20], shadow memory [26], pool allocation [11].

Another way to reduce overhead due to runtime checks is to make use of precise static analysis. The results from such analysis could be used towards removing 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. 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.

The goal of this paper is advance the 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.

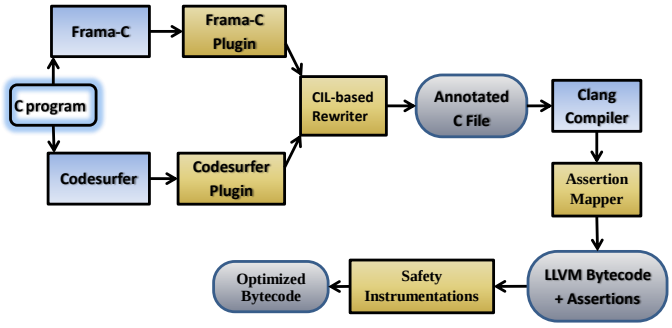

Figure 1: Implementation Architecture.

```
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

3 Design and Implementation

The strategy we designed involves a series of steps, listed here and described in detail below:

- *Deep Analysis* Information about variable values and program sequence is extracted using the external tools, Frama-C and Codesurfer; converted into a common semantics; and prepared for injection into the target C source code. This sequence is represented in the first six boxes in Figure 1.
- *Annotation Capture* Annotations prepared for injection are captured as string assignments and inserted in the C source code — the 'Annotated C-file' box in the figure — for processing by the Clang Compiler.
- *Transport to the Backend* Assertions prepared as string assignments are converted by the 'Annotation Mapper' to metadata, to transport assertions associated with the target variables to the compiler backend.
- *Check Elimination* During the passes that insert 'Runtime Instrumentation,' metadata analysis enables removal of unnecessary checks.

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 cross 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.

Running Example. To illustrate our approach, we provide a simple running example in Listing 1 containing several operations that will be instrumented with run time checks. In reality, these run time checks are inserted in the intermediate code generated by the front end, however, we show them in the

```
1  if(shadow(i)!=0) throw_UAF_Exception();
2  while(i<100) {
3    if(i<= 0 || i >= sizeof(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 from the running example with runtime checks

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 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 run time check, inserted by Safecode. The check verifies that the access is within the bounds of the array `p` and throws an exception if this is not true. Finally, Lines 6-8 contain the code transformed by the Integer Overflow Check instrumentation. In this code, the original multiplication is substituted with a safe multiplication version, which returns a structure containing the result and a flag indicating if overflow occurred.

As shown by the example, the runtime checks inserted in Listing 2 by the safety instrumentations are not necessary. In fact, the value of variable `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.

Evidently, if we know in advance the value range and memory safety information of the variables at compilation time, we can remove these checks. Our framework transports such information from static analysis tools to MESCs, improving the performance of target programs by removing these unnecessary runtime checks.

3.1 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 run time checks. To remove runtime checks dedicated to catching *out of bounds* memory accesses and *use after free* bugs, we use as external analyzers two tools, Frama-C and Codesurfer [9][2][28][1]. These tools exhibit several advantages with respect to LLVM's analysis capabilities. Both tools are able to perform whole program analysis, spanning multiple compilation units and procedures. In addition, they are not as constrained as LLVM with respect to performance, and thus can perform a deeper and more complex analysis.

Frama-C. This is an analysis framework for C programs that can be extended with different plugins for

different types of analysis. Among these plugins, one of the most widely used is the *value range analysis*. After building an internal representation for a program, this plugin is able, through abstract interpretation, to derive the ranges of the variable values. The results of Frama-C's value analysis are guaranteed to be sound [9]. For instance, in Listing 1, Frama-C is able to determine that the range of the index `i` is between 0 and 100. 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 that an integer overflow check related to the multiplication can be removed.

Although the value-range information of variables is computed by the Frama-C Value Analysis plugin, 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, that visits every instruction in the AST tree inside Frama-C, extracts all the variables at each node, and queries the value analysis plug-in for each variable to get the corresponding value ranges. The results are then stored in a file to be used later by the CIL-based rewriter, described below.

CodeSurfer. This framework provides different types of facilities for analyzing full programs. 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 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.

CodeSurfer provides several advanced analysis and query capabilities, as well as an API to build plugins for customized queries. Among these queries, *Backward program slicing* with respect to a program point A retrieves all the program points that can influence the values of the variables used in A. *Forward program slicing* with respect to a program point A retrieves all the program points that can be influenced by A. CodeSurfer also provides pointer and alias analysis capabilities and is able to discover pointer aliases and to include the effects of aliasing in the analysis.

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 implies there exists 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, on the other hand, 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 run time check associated with that use can be removed.

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

1. Issue backward and forward slice queries to find the order of execution among statements containing a call to a `free` and statements that use the pointer being freed. More specifically, 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 (e.g., they are both inside a loop body).
2. Find the program points that contain variable uses that are not related to a `free(p)` statement. The latter include all those uses for which runtime checks are inserted by Address Sanitizer, but which are

not usually free-d 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.

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

Information granularity. One of the main issues in using external analyzers for removing checks is the granularity gap between the operations and information produced by these analyzers and the operations of the safety instrumentation frameworks. The latter usually operate at a lower code abstraction level (intermediate representation) by instrumenting memory accesses at that level. The former typically operate at a more abstract level (e.g., source code) and provide analysis information at that level. For instance, the Safecode compiler pass instruments with run time checks the LLVM IR `load`, `store`, and `gep` operations (which are used for array accesses), while the Frama-C analyzer, when used in relation to Safecode, performs abstract interpretation on the C source code and computes the variables' value ranges. As another example, Address Sanitizer instruments with run time checks LLVM IR `load` and `store` operations, while Codesurfer builds a System Dependency Graph to represent data and control dependencies between the program points in the source code.

In the presence of this gap, the first task in the analysis is to retrieve and transform the information produced by the external analyzers into a form suitable for being passed to the next steps. This implies fixing the granularity of the information to an intermediate level between the source code and the IR code and retrieving or transforming the analysis information at that level. In our approach, we fix the granularity to the *uses* and *definitions* of single source variables and thus retrieve from the analyzers information about variable uses and definitions. The advantage of this choice lies in the fact that these uses are available to the instrumentation framework of the compiler, and can be easily inferred by its analyzers.

Analysis soundness. Another issue in using external static analyzers is that, in general, they may employ 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. Therefore, if an analyzer cannot infer the information needed to remove a check with 100% certainty, we do not include that information in the steps that follow. The practical effect of this choice is that a significant percentage of run time checks may not be removed.

3.2 Annotation Capture

To provide a common framework for both the `out of bounds` and `use after free` optimizations, we specify a common assertion language to express value-range and memory safety information about variables in each program location. The syntax of our specification is described in Table 2. 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 represent the value range and (2) the memory safety of a specific variable. The next challenge is to transport these assertions to the compiler frontend. As our framework is designed to use different analysis tools, we need to normalize the code so that the source location will be consistent for later injection. To this end, we leverage a transformation tool to perform normalization and transformation. The challenge of transformation is that the injected assertions must not interfere with the semantics of the program.

Our solution is to store the assertions in string variables, which are specially named to avoid interference with the existing program variables. These string variables are injected before the corresponding instructions in source code, and are propagated to the compiler back-end through the standard code generation phase of the Clang frontend.

<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 2: Syntax of the common assertion language

To inject assertions as string variables into source code, we have implemented a transformation tool, which is based on CIL (C Intermediate Language) [24]. 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 plug-in takes as input a C source file and the corresponding set of assertion specifications, and visits all instructions in that C source file. If an instruction matches the variable name and file location of an annotation, the annotation is injected before the instruction in a dummy string variable. Listing 3 demonstrates the output of our transformation tool that injects assertions in string variables in corresponding source locations.

3.3 Transport to the Backend

Once the information about variable uses and definitions is produced by the external analyzers, the goal of the next step is to transport this information to the backend. 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 SSA nature of the LLVM 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 issue 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 3).

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 3.2.

Assertion Mapping. 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 might otherwise be removed by the optimization passes as dead code. The *Assertion Mapper* pass works according to the steps described

```
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 3: Program from the running example with injected assertions

```
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 4: LLVM IR code compiled from the running example

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 4, 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 5. For every LLVM instruction, the text after the `!` shows the safecode-related metadata. These contain the range

```
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 5: LLVM IR Annotated with Assertion Metadata

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 index of the array, and the metadata associated to line 5 provides the size of the array.

The metadata related to the Address Sanitizer's check are similarly attached to `load` and `store` statements. These 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 5, 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 operations and sign extension operations.

3.4 Backend Traversal

Once the assertions are available to the backend, 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 necessarily modifies the mappings between source and LLVM variables discovered in the previous step, thus invalidating the assertions.

To deal with this issue, we 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. For instance, in code Listing 4, all the instructions are associated with a copy of the range of variable `i`. In addition, when possible, we compute new assertions by using existing ones. For instance, in Listing 4, if the range of the variable `%i` is known, the range of the multiplication result in Line 3, is computed from the range of `i`, as described above in Section 3.3.

3.5 Check Elimination

When the assertions reach the instrumentation passes, the information they contain determines if a check is needed or not. 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 Check Removal Implementation. 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 run time check.

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 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: 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. 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 implementation. The implementation of the integer overflow checks removal is very similar to that of Safecode. In particular, we intercept every arithmetic operation that is instrumented by the pass 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 do not insert the check if all the values of the range fall below that maximum integer

Use after free check removal implementation. 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. If a memory access is associated with an assertion that claims that it is safe, we skip the run time check insertion. For instance, Listing 6 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 6: Optimized program after leveraging assertions

4 Evaluation

4.1 Methods

This work incorporated the "acslgen" plugin written for the frama-C framework during previous work in the lab [14]. This plugin leverages the "value analysis" plugin to generate value constraints on variables, converting these constraints to annotations in the source code. The modified source code is then compiled using a customized version of LLVM, which applies the annotations to variable-attached metadata that act as "witness" to code transformations, to ensure that constraint annotations are properly transformed to the appropriate values in the code of the compiler's intermediate representation (in which variable names are modified) [22]. In this way, variable constraints are carried across the compilation procedure to be applied during the compiler passes selected to be improved, explicitly eliminating instances of security checks against variable manipulations in locations where the variable constraints prove that they will not be needed. This protocol mitigates the performance overhead due to targeted compiler passes that add runtime security checks.

Benchmark	**Description**	**Line Count**	**Annotation Count**	**% 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 3: Benchmark Source Size and Annotation Coverage

The "safecode" pass generates several protections to enhance memory safety[2]:

1. Pointers are restricted to declared array sizes by checking references to array bounds.
2. Loads and stores of memory objects are guaranteed to be valid.
3. Type safety is guaranteed for proven type-safe objects.
4. Dereferencing of dangling pointers is managed under sound operational semantics.
5. Dangling pointers can optionally be detected, at the cost of additional overhead

The "integer overflow check" pass inserts checks at every arithmetic operation to determine if the the operation resulted in overflow and was followed by an undefined usage of the result [12]. Instances of this check can be eliminated when value analysis shows that overflow cannot occur.

We compared performance of the three executables: the original code, the code with all runtime security checks, and the code with runtime security checks judiciously removed where they were determined to be unnecessary. Tests were run separately for removal of safecode checks and for removal of integer overflow checks.

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

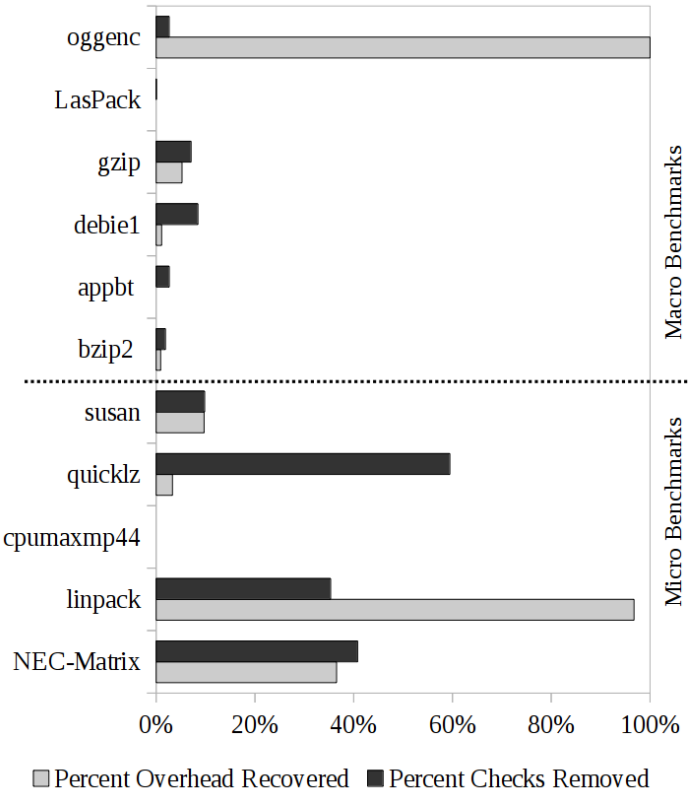

Figure 2: Safecode Benchmarks Performance

Execution for a selected set of benchmarks was examined. Standard benchmarks required source code modifications:

1. Apply timing instrumentation
   At entry to main, capture user time, and on exit, again capture user time and output the difference in microseconds.
2. Apply hand corrections in locations where changes introduced by frama-C interfered with building the benchmark package.
   Certain constructions generated duplicate labels; where annotations were added within case instances, braces were needed to prevent compilation errors.
3. frama-C build does not always perform where original programs required complex makefiles; these cases were fixed by build restructuring.

We selected a range of benchmarks for testing the efficacy of the procedure, as listed in Table 3. Criteria for selecting benchmarks included

1. Large non-commentary lines of code count with small number of source files
2. Significant number (at least several hundreds) of non-fixed array references
3. Successful frama-C build
   The frama-C utility does not yet support makefile builds, so some benchmarks could not be included in the study.

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

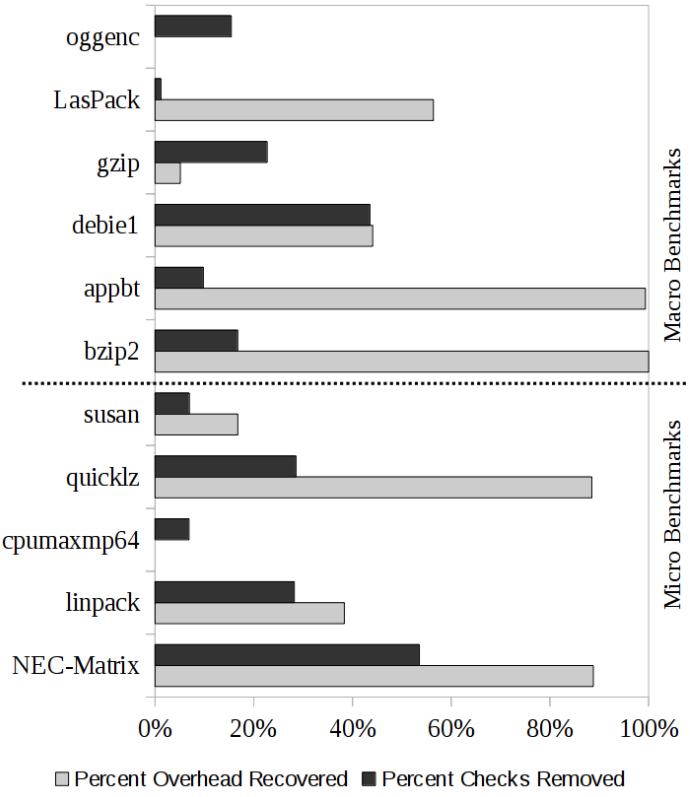

Figure 3: Integer Overflow Check Benchmark Performance

course of its execution. Each program was run at least ten times and the times were averaged. Experiments were run using LLVM versions 3.4 and 3.2, Frama-C version Fluorine-20130601. 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.

4.2 Benchmarks

We selected 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 as reported by the LINUX tool *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; complicated makefile builds not supported by Frama-C were avoided.

Table 3 illustrates the source code line counts and the percentage of source lines that were annotated by Frama-C, using value analysis to establish variable range constraints. 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.

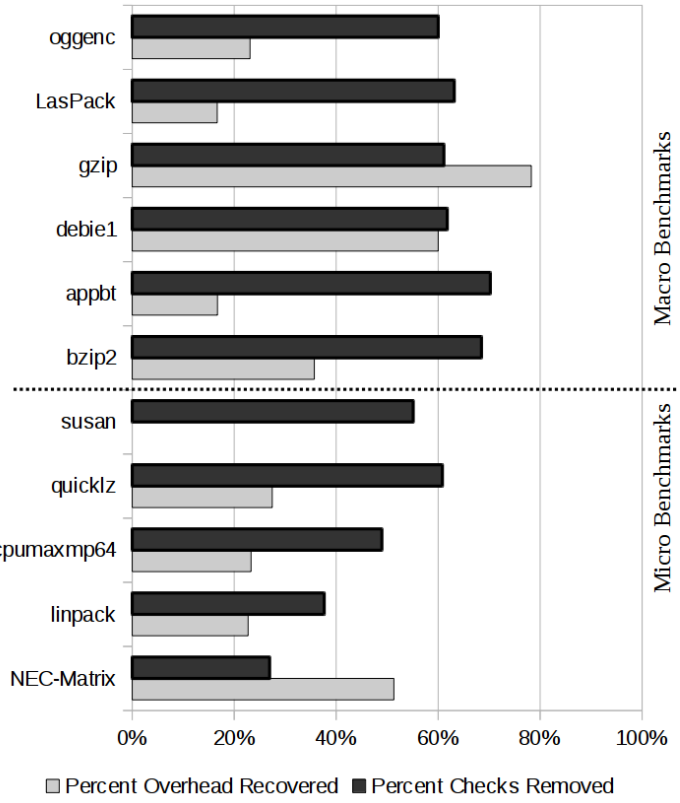

Figure 4: Address Check Benchmark Performance

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.

4.3 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 largest to smallest. Two metrics are given in the charts: the percentage of runtime checks that were *removed* by our protocol, and the percentage of the performance overhead due to runtime checks that was recovered by the framework:

$$\%\text{Overhead\_Recovered} \;=\; 1 \;-\; \frac{\text{Some\_Checks\_Removed} \;-\; \text{Original}}{\text{All\_Checks\_Present} \;-\; \text{Original}} \tag{1}$$

Safecode. Figure 2 shows results following removal of safecode checks. This class of runtime checks had the most severe effect on performance overhead, as seen above in Table 1. 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 3), 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. 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 4. As can be noted, the improvements in the run time overhead of Address Sanitizer range between 15-40%. In the instance of the `susan` benchmark, runtime memory safety errors prevented running to completion.

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.

Benchmark	**Safecode**	**IOC**	**Address**
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	
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 4: Net Benchmark Overhead Following Check Elimination

Summary.

Results of our benchmark experiments are encouraging, although varying widely with the programs and sanitizations being tested. Our best runs show recovery of more than half the overhead due to runtime security checks. Net overhead for all the experiments is seen in Table 4, where results in each case are presented as the overhead factor with all checks in place, followed by an arrow, followed by the overhead factor after some checks were removed by our protocols. This presentation is more revealing than the charts in Figures 2, 2, and 2, which only illustrate the relative improvements due to checks removal. Table 4 gives the absolute overheads running the optimized programs, compared with the performance of the original, unsafe code. We are in the process of analyzing these results to seek a better understanding of the difference in value for the protocol in different contexts; we are also investigating ways to assess the potential a given program has for improvement, so that the effort in optimizing its safety instrumentation is effectively spent.

5 Discussion

The present work expands on the findings of the earlier paper [14], in evaluating the applicability of static analysis research tools to the optimization of runtime security checks. We find that larger benchmarks present difficulties in building under frama-C that will require tool enhancements to overcome, before this protocol can be widely accepted. So far we see less reliable improvement in "safecode" pass improvements than hoped for, even in cases where significant check reductions occur; for the quicklz benchmark, a 59% reduction in checks only yielded a 4% performance improvement. However, additional work may show better results with larger benchmarks (pending).

As seen in our earlier work, runtime improvement varies greatly with the application. Looking at the integer overflow checks, while quicklz exhibits a similar improvement in performance as the percentage of checks removed, oggenc exhibits about twice the performance boost compared to the percentage of checks removed, and for LasPack, in which only 1% of the checks were removed (6 out of 490), the performance recovery is nearly 30%. This sporadic success is not unexpected, due to the random nature of identifiable value ranges that are discovered by frama-C, which may occur within initialization code executed only once, or in heavily traveled functions. Performance overhead reduction reflects this manifestation.

6 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 [10, 26, 17, 20, 21, 30, 3, 8, 31, 7]. 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 ([26]), for example, uses shadow memory, which computes the location of the status information very quickly; other approaches incorporate different efficient data structures ([4]). 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 [29]. However, this system makes no safety guarantees, and it may remove checks which are necessary for safety.

[22, 14] tackle the problem of propagating assertions in a compiler. [22] develops the theory using refinement relations and [14] 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.

DangNull instruments 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 [17].

Other techniques rely on changing the memory allocation layouts, so that memory safety errors do not occur, or occur with low likelihood [6, 3, 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 [6, 3]. 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 [25] and Electric Fence [15] also have a very high overhead both in memory and running time.

Dietz *et al*[12] provide a thorough analysis of integer overflow issues, causes, and justifications and risks as an intentional coding method. Necula *et al*[23] analyze deployed C programs and find that nearly 90% of pointers can be proven to be memory safe; their CCured framework can target the unsafe ones in a way conceptually similar to our approach. The work in our lab emphasizes the usefulness of determining constraints on variable values, which inform the annotations that are converted to compiler metadata.

7 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.

References

[1] CodeSurfer. http://www.grammatech.com/research/technologies/codesurfer.

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

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

[4] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *18th Conference on USENIX Security Symposium (SSYM'09)*, pages 51–66. USENIX Association, 2009.

[5] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, pages 14–25, 2004.

[6] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. *SIGPLAN Not.*, 41(6):158–168, June 2006.

[7] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2011)*, pages 213–223. IEEE Computer Society, 2011.

[8] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *International Symposium on Software Testing and Analysis (ISSTA 2012)*, pages 133–143. ACM.

[9] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. In *10th International Conference on Software Engineering and Formal Methods (SEFM 2012)*, pages 233–247. Springer-Verlag, 2012.

[10] Dinakar Dhurjati and Vikram Adve. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. Technical report, Shanghai, China, May 2006.

[11] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. In *2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems(LCTES 2003)*, pages 69–80. ACM.

[12] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in c/c++. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 760–770, Piscataway, NJ, USA, 2012. IEEE Press.

[13] Vijay DSilva, Mathias Payer, and Dawn Song. The Correctness-Security Gap in Compiler Optimization – LangSec'15. In *LangSec'15, IEEE Symposium on Security and Privacy Workshops*, San Jose, CA, 2015.

[14] Rigel Gjomemo, Kedar S. Namjoshi, Phu H. Phung, V.N. Venkatakrishnan, and Lenore D. Zuck. From verification to optimizations. D'Souza, Deepak (ed.) et al., Verification, model checking, and abstract interpretation. 16th international conference, VMCAI 2015, Mumbai, India, January 12–14, 2015. Proceedings. Berlin: Springer. Lecture Notes in Computer Science 8931, 300-317 (2015)., 2015.

[15] N. Joukov, A. Kashyap, G. Sivathanu, and E. Zadok. Kefence: An electric fence for kernel buffers. In *First ACM Workshop on Storage Security and Survivability (StorageSS 2005)*, pages 37–43, FairFax, VA, November 2005. ACM.

[16] Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve. Ensuring code safety without runtime checks for real-time control systems. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '02, pages 288–297, New York, NY, USA, 2002. ACM.

[17] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015*. The Internet Society, 2015.

[18] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54. ACM, 2006.

[19] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[20] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2009)*, pages 245–258. ACM.

[21] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 31–40, New York, NY, USA, 2010. ACM.

[22] Kedar S Namjoshi and Lenore D Zuck. Witnessing program transformations. In Francesco Logozzo and Manuel Fahndrich, editors, *Static Analysis*, volume 7935 of *Lecture Notes in Computer Science*, pages 304–323. Springer Berlin Heidelberg, 2013.

[23] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[24] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *11th International Conference on Compiler Construction (CC)*, pages 213–228. Springer-Verlag, 2002.

[25] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 89–100. ACM.

[26] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Conference on Annual Technical Conference (USENIX ATC 2012)*, pages 28–28. USENIX Association.

[27] L. Szekeres, M. Payer, Tao Wei, and D. Song. Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62, May 2013.

[28] Tim Teitelbaum. Codesurfer. *ACM SIGSOFT Software Engineering Notes*, 25(1), 2000.

[29] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High System-Code Security with Low Overhead. In *36th IEEE Symposium on Security and Privacy*, 2015.

[30] Wei Xu, Daniel 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)*, pages 117–126. ACM.

[31] Mingwei Zhang and R. Sekar. Control flow integrity for cots binaries. In *22Nd USENIX Conference on Security*, SEC'13, pages 337–352. USENIX Association, 2013.

[32] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formal verification of ssa-based optimizations for LLVM. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, pages 175–186. ACM.