RV'02 Preliminary Version

Reducing the Overhead of Dynamic Analysis [1]

Suan Hsi Yong [2] and Susan Horwitz [3]

Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison, WI 53706 USA

Abstract

Dynamic analysis (instrumenting programs with code to detect and prevent errors during program execution) can be an effective approach to debugging, as well as an effective means to prevent harm being caused by malicious code. One problem with this approach is the runtime overhead introduced by the instrumentation. We define several techniques that involve using the results of static analysis to identify some cases where instrumentation can safely be removed. While we have designed the techniques with a specific dynamic analysis in mind (that used by the Runtime Type-Checking tool), the ideas may be of more general applicability.

1 Introduction

Languages like C and C++ that allow potentially unsafe operations such as pointer arithmetic, casting, and explicit memory management open the door to many difficult-to-detect errors as well as providing opportunities for malicious code to be inserted into programs [20]. To address these problems, a number of systems have been developed that involve dynamic analysis: instrumenting a program so that errors like out-of-bounds array indexes and bad pointer dereferences are detected when they occur during execution [2,7,14,13,9,10]. In some cases, these kinds of dynamic checks are even mandated by the language definition (e.g., Java guarantees that an exception will be thrown whenever an index is out of bounds, or a bad cast is performed).

Naturally, the benefits of dynamic analysis have an associated cost: the instrumentation introduces a certain amount of runtime overhead. This paper proposes several techniques for reducing the overhead by using static analysis to identify some cases in which instrumentation can safely be omitted.

[1] This work was supported in part by the National Science Foundation under grants CCR-9970707 and CCR-9987435.

[2] Email: suan@cs.wisc.edu

[3] Email: horwitz@cs.wisc.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

While the techniques were designed for one particular tool: the Runtime Type-Checking (RTC) tool [10], the ideas may be of more general applicability.

The remainder of the paper is organized as follows. Section 2 provides background on the RTC tool; Section 3 describes several static analyses that can be used to identify unnecessary instrumentation: Section 3.2 describes an approach to classify expressions into type-safety levels, so that instrumentation for "type-safe" expressions can be eliminated, and Section 3.3 presents three refinements to this analysis (one to account for uses of uninitialized data, one to remove some checks for null-pointer dereferences, and one to identify redundant instrumentation). Section 4 presents experimental results that help gauge the potential benefit of these optimizations, Section 5 discusses related work, and Section 6 concludes.

2 The RTC Tool

The RTC (Runtime Type-Checking) tool instruments C programs so that the runtime type of every memory location is tracked during program execution, and inconsistent type uses are reported as warnings and errors. Whenever a value v is written into a location l, l's runtime type is updated with v's runtime type. Also, this runtime type is compared with l's declared type: if they do not match, a *warning* message is issued (a warning message is an indication that unusual behavior has been observed, and may be useful for diagnosing the root cause of a later error). Whenever the value in a location is used, its runtime type is checked, and if the type is inappropriate in the context in which the value is being used, an *error* message is issued; to avoid cascading error messages, the runtime type is set to the correct type after an error message is generated.

2.1 Motivating Example

While a number of other tools have been proposed to detect out-of-bounds array accesses and bad pointer dereferences, the type-checking approach of the RTC tool can also detect more subtle errors involving type misuses. One such class of errors has to do with a programming style in which C programmers simulate classes and inheritance using structures [16]. For example, the following declarations might be used to simulate the declaration of a superclass `Base` and a subclass `Sub`:

```
struct Base { int a1; int *a2; };
struct Sub { int b1; int *b2; char b3; };
```

A function might be written to perform some operation on objects of the superclass:

```
void f ( struct Base *b ) {
   b->a1 = ...
   b->a2 = ...
}
```

and the function might be called with actual arguments either of type `struct Base *` or `struct Sub *`:

```
struct Base base;
struct Sub sub;
f(&base);
f(&sub);
```

The ANSI C standard guarantees that the first field of every structure is stored at offset 0, and that if two structures have a common initial sequence — an initial sequence of one or more fields with compatible types — then corresponding fields in that initial sequence are stored at the same offsets. Thus, in this example, fields `a1` and `b1` are both guaranteed to be at offset 0, and fields `a2` and `b2` are both guaranteed to be at the same offset. Therefore, while the second call, `f(&sub)`, would cause a compile-time warning (which could be averted with an appropriate type cast), it would cause neither a compile-time error nor a runtime error, and the assignments in function `f` would correctly set the values of `sub.b1` and `sub.b2`.

However, the programmer might forget the convention that `struct Sub` is supposed to be a subtype of `struct Base`, and while making changes to the code might change the type of one of the common fields, add a new field to `struct Base` without adding the same field to `struct Sub`, or add a new field to `struct Sub` before field `b2`. For example, suppose a new `int` field, `i1` is added to `struct Sub`:

```
struct Sub { int b1; int i1; int *b2; char b3; };
```

Now, when the second call to `f` is executed, the assignment `b->a2 = ...` would write into the `i1` field of `sub` rather than the `b2` field. The fact that the `b2` field is not correctly set by the call to `f`, or that the `i1` field is overwritten with an unintended value, will probably either lead to a runtime error later in the execution, or cause the program to produce incorrect output.

The tracking of runtime types performed by the RTC tool can help the programmer uncover the source of this logical error. The assignment `b->a2 = ...` causes `sub.i1` to be tagged with type *pointer*. A later use of `sub.i1` in a context that requires an *int* would result in an error message due to the mismatch between the required type (*int*) and the current runtime type (*pointer*).

Note that in this example, a tool like Purify [7] would not report any errors, because there are no bad pointer or array accesses: function `f` is not writing outside the bounds of its structure parameter, it just happens to be the wrong part of that structure from the programmer's point of view.

2.2 Tracking Types

The RTC tool associates with each memory location a runtime type represented as a tuple $\langle \sigma, \mathit{size} \rangle$, where σ is one of *unallocated*, *uninitialized*, *zero*, *integral*, *real*, and *pointer*, and *size* is the size (in bytes) of the type. For example, a `char` is represented by $\langle \mathit{integral}, 1 \rangle$, and a `float` by $\langle \mathit{real}, 4 \rangle$ (on most platforms). Pointers to different types are represented by the same runtime type, $\langle \mathit{pointer}, 4 \rangle$ (on a platform where all pointers are 4 bytes in size), and `typedef`s are not treated as separate types. For aggregate objects (structures and arrays), the runtime type of each field/element is tracked separately. The special *zero* type is used for memory locations that are assigned the literal 0; it is treated as being compatible with all C types. The runtime types are stored in a "mirror" of the memory used by the program, with each byte of memory mapped to a four-bit nibble in the mirror (thus incurring a 50% space overhead).

The RTC tool has been implemented to handle all of ANSI C. It translates a given set of preprocessed C source files into instrumented C files. These are then compiled and linked with the RTC library, producing an executable that performs runtime type checking and reports error and warning messages.

The instrumentation phase is a source-to-source translation of the C program; it performs a syntax-directed transformation on the program's abstract-syntax tree to add calls to RTC library functions that track the runtime types. The operations performed by these library functions can be grouped into the following classifications:

declare - a variable declaration is instrumented to set the runtime type in the variable's mirror to *uninitialized*. (Initially, the mirror for all memory is tagged *unallocated*.)

verify - a use of a memory location x in the context of a type τ is instrumented to compare the runtime type in the mirror of x with τ. If the types are not compatible, an error message is issued, and the runtime type of x is corrected to τ (to prevent cascading error messages).

verify-pointer - a pointer dereference is instrumented to check whether the mirror of the pointer's target is *unallocated*. If it is, an error message is issued. This check detects dangling pointer dereferences, dereferences of certain stray pointers (those that point between or beyond allocated blocks), and also null-pointer dereferences (because the mirror of memory location 0 is tagged *unallocated*).

copy - an assignment statement is instrumented to copy the runtime type of the right-hand-side value into the mirror of the left-hand-side location; additionally, if the runtime type of the assigned value does not match the static type of the assignment, a warning message is issued.

The tool is designed so that instrumented modules can be linked with uninstrumented ones. This flexibility is useful if, for example, a programmer only wants to debug one small component of a large program: they can instrument only the files of interest, and link them with the remaining uninstrumented object modules. A caveat when doing this, however, is that it may lead to spurious warning and error messages because the uninstrumented parts of the code do not maintain the necessary runtime type information for the memory locations they use. For example, if a reference to a valid object in the uninstrumented portion of the program is passed to an instrumented function, the tool will consider that object unallocated, and may output a spurious error message if that object is referenced.

This problem extends, in general, to library modules. For example, the flow of values in a function like `memcpy`, the initialization of values from input in a function like `fgets`, and the types of the data in a static buffer returned by a function like `ctime` would not be captured. To handle these, we have created a collection of instrumented versions of common library functions that affect type flow. These are wrappers of the original functions, hand-written to perform the necessary tag-update operations in the RTC mirror to capture their type behavior.

Included among these instrumented library functions are memory-management functions. Each call to `malloc` (or one of its relatives) is replaced with a call to a wrapper version which, upon successfully allocating a block of memory, sets the mirror for that memory block to *uninitialized*. Similarly, the wrapper version of the `free` function resets the mirror to *unallocated*. The `malloc` wrapper also adds padding between allocated blocks to decrease the likelihood of a stray pointer jumping from one block to another (this is the approach used by Purify [7]).

The RTC tool was able to detect bugs in some SPEC benchmarks (`go`, `ijpeg`), Solaris utilities (`nroff`, `col`, etc.), and Olden benchmarks (`health`, `voronoi`) [10]. Most of the errors were out-of-bounds array or pointer accesses. In the Solaris utilities, the out-of-bounds accesses resulted in program crashes; in the SPEC cases, the errors had no apparent effect on the execution, which made the errors difficult to detect without the use of a tool like the RTC tool. In every case, the RTC tool was able to detect the out-of-bounds memory accesses because the type of the pointed-to memory was different from the expected type.

Finally, the RTC tool lends itself naturally to interactive debugging. When a warning or error message is issued, a signal (`SIGUSR1`) is sent, and can be intercepted by an interactive debugger like GDB [17]. The user can then examine memory locations, including the mirror, and make use of GDB's features to help track down the cause of an error.

3 Eliminating Unnecessary Checks

While the initial implementation of the RTC tool demonstrated its ability to find errors in real programs as described above, a shortcoming of that implementation was poor performance: in the worst case, an instrumented program ran 130 times slower than the non-instrumented version. This is because the RTC tool instruments *every* expression in the program and tracks the runtime type of *every* memory location used in the program.

Outlined below are some strategies for reducing the runtime overhead by using the results of static analysis to identify and remove unnecessary instrumentation. First, we describe a flow-insensitive analysis that identifies "type-safe" expressions that need no instrumentation. Next, we describe three flow-sensitive refinements. These analyses have not yet been fully implemented; to gauge the potential speedup that they will provide, we have implemented a simpler analysis and tried it on a number of programs. The results of those experiments are reported in Section 4.

3.1 Assumptions

The static analyses described below require the results of pointer analysis to account for possible aliasing in the program. We assume that a (flow-insensitive) points-to analysis (e.g., [1,21,18,5]) has been performed, so that each pointer p in the program is associated with a points-to set, containing variables to which p may point at some point in the program.

We also assume that the assignment statements in the input program have been normalized to the forms defined by the following context-free grammar:

$$\begin{array}{rcl} \mathit{assign} & \Rightarrow & \mathit{lvalue} = \mathit{rvalue} \\ & | & \mathit{lvalue} = (\tau)_{cvt}\mathit{rvalue} \\ & | & \mathit{lvalue} = (\tau)_{ext}\mathit{rvalue} \\ & | & \mathit{lvalue} = (\tau)_{cpy}\mathit{rvalue} \\ \mathit{lvalue} & \Rightarrow & \mathit{var} \mid *\mathit{var} \\ \mathit{rvalue} & \Rightarrow & \mathit{const} \mid \mathit{var} \mid *\mathit{var} \mid \texttt{\&}\mathit{var} \mid \mathit{var} \oplus \mathit{var} \end{array}$$

where *const* is a constant, *var* is a variable, and $\oplus$ represents any C binary operator. Type-casts are divided into three forms. The first form, $(\tau)_{cvt}e$, is a type cast that involves a change in representation, and includes conversions (e.g., between integers and floating-point values) and truncation of data (e.g., when type-casting a `long int` into a `short int`). The second form, $(\tau)_{ext}e$, represents type-casts that extend data from a smaller type to a larger type with no change in the data bits (e.g., from a `short int` into a `long int`). The third form, $(\tau)_{cpy}e$, represents type-casts where there is no change in the form of the data, and includes casts between pointers and integers (of the same size). The difference between these forms that concerns us is that, for $(\tau)_{cvt}e$, the RTC instrumentation checks that the runtime type of e is compatible with its static type; if they are incompatible, an error message is issued, and the runtime type of the expression is set to τ to suppress further error messages. For $(\tau)_{ext}e$ and $(\tau)_{cpy}e$, no such check and correction is performed.

Most C assignment statements can be normalized as defined above. For example, an assignment that involves an array index, such as $x = a[i]$, can be rewritten as $tmp = a + i;\ x = *tmp$. Details of how to handle the remaining

C constructs (e.g., structures, unions, and function calls) remain to be worked out.

3.2 Type-Safety-Level Analysis

Our first static analysis is a flow-insensitive type-safety-level analysis that partitions the expressions in a program into levels of "type safety", so that certain classes of runtime instrumentation (see Section 2.2) can be eliminated for expressions at certain type-safety levels.

The proposed approach classifies expressions in the program into the following type-safety levels:

safe - An expression whose runtime type is guaranteed always to be compatible with its static type, and for which all instrumentation can be eliminated.

unsafe - An expression whose runtime type may be incompatible with its static type; this includes expressions of the form `*p`, when the pointer `p` may be NULL, or may contain an invalid address. An *unsafe* expression must be fully instrumented.

tracked - An l-value expression whose runtime type is always compatible with its static type, but which may be pointed to by an unsafe pointer or by a pointer whose points-to set includes a location with an incompatible type. A tracked expression's corresponding location needs to have its runtime type initialized (to its static type) in the mirror, but instrumentation for verifying and copying the runtime type for a *tracked* expression can be eliminated.

Figure 1 presents an example code fragment to illustrate the intuition behind the proposed approach. Since the approach is flow-insensitive, the order of the statements is ignored in the analysis. The expressions `p0`, `p1` and `p2` are *safe* because they are only assigned pointer-typed values (recall that the RTC tool does not differentiate between pointer types, so the fact that `p1` is assigned both the address of an `int` variable and the address of a `float` variable is not important; also recall that the literal 0 is treated as being compatible with all types, including pointers).

The expressions `f`, `*p0`, `*p1`, and `*p2` are all *unsafe*. Variable `f` is *unsafe* because the assignment at line 13 could write an `int` value into `f` via `*p1`. The expression `*p0` is *unsafe* because `p0` may be NULL (due to the assignment at line 6); `*p1` and `*p2` are *unsafe* because while they each have a static type of `int *`, `*p1` may refer to a `float` (because of the assignment at line 10), and `*p2` may refer to an invalid address (because of the pointer arithmetic at line 11).

Finally, the expression `i` is *tracked*. Although `i` will always contain an `int` value, it may be pointed to by `p1`, which also includes `f` — a `float` variable — in its points-to set. This means that every use of `*p1` will be instrumented to check its runtime type, and so every location in `p1`'s points-to set — including

Code		Expression	Type-safety level
1.	`int i;`	`p0`, `p1`, `p2`	*safe*
2.	`int *p0,*p1,*p2;`	`f`, `*p0`, `*p1`, `*p2`	*unsafe*
3.	`float f;`	`i`	*tracked*
4.	`i = 1;`		
5.	`f = 2.3;`		
6.	`p0 = 0;`		
7.	`if(*p0 == 0)`		
8.	`  p1 = &i;`		
9.	`else`		
10.	`  p1 = (int *) &f;`		
11.	`p2 = p1 + i;`		
12.	`if(*p2 != 0)`		
13.	`  *p1 = 4;`		

Fig. 1. Type-safety example.

`i` — must have its runtime type recorded in the mirror.

Within this framework, we can devise schemes of varying precision to determine the type-safety level of each expression. Using the following ordering,

$$unsafe < tracked < safe$$

any scheme that classifies each expression at a level less than or equal to its true level is a safe approximation. For example, the unoptimized RTC tool corresponds to one extreme, where all expressions are considered *unsafe*. The next three sections describe an efficient flow-insensitive analysis to classify the type-safety of expressions. The analysis works as follows:

Step 1: Build an *assignment graph* in which the nodes represent the expressions in the program, and the edges represent the flow of runtime types due to assignments.

Step 2: Compute a *runtime-type* attribute for each node in the graph.

Step 3: Compute the type-safety level for each node in the graph (and thus for each expression in the program).

3.2.1 Step 1: Building the assignment graph

The first step of the analysis involves building an assignment graph that records the flow of runtime types among the expressions in the program. Each node in the assignment graph corresponds to an expression, and represents an "abstract object" of one of four forms: v, $*v$, $\oplus_\tau v$, and $\textsc{value}_\tau$. The v node represents a variable v, $*v$ represents a dereference, $\oplus_\tau v$ represents an arithmetic operation on v (resulting in a value of static type τ), and $\textsc{value}_\tau$ represents a value of type τ, e.g., from a constant expression. For both $\oplus_\tau v$, and $\textsc{value}_\tau$, τ will be either a scalar C type: *char*, *int*, *float*, etc, or one of

expr	*AbsObj*(*expr*)
0	$\{\text{VALUE}_{zero}\}$
C_τ [1]	$\{\text{VALUE}_\tau\}$
$\&y$	$\{\text{VALUE}_{valid-ptr}\}$
y	$\{y\}$
$*y$	$\{*y\}$
$y \oplus z$ [2]	$\{\oplus_\tau y, \oplus_\tau z\}$

[1] C_τ is a non-zero constant of type τ
[2] The expression $y \oplus z$ has static type τ

(a)

Assignment	Edge(s) in Graph
$e_1 = e_2$	$n_1 \xleftarrow{=} n_2,\quad n_1 \in AbsObj(e_1),\ n_2 \in AbsObj(e_2)$
$e_1 = (\tau)_{cpy}e_2$	$n_1 \xleftarrow{=} n_2,\quad n_1 \in AbsObj(e_1),\ n_2 \in AbsObj(e_2)$
$e_1 = (\tau)_{cvt}e_2$	$n_1 \xleftarrow{cvt} n_2,\quad n_1 \in AbsObj(e_1),\ n_2 \in AbsObj(e_2)$
$e_1 = (\tau)_{ext}e_2$	$n_1 \xleftarrow{ext} n_2,\quad n_1 \in AbsObj(e_1),\ n_2 \in AbsObj(e_2)$

(b)

Fig. 2. Rules for initializing the assignment graph.

the following special types:

***valid-ptr*:** A pointer expression that is guaranteed to evaluate to a valid address (the address of an allocated memory location) has type *valid-ptr*. For example, the expression `&x` has type *valid-ptr*.

***pointer*:** A pointer expression that may evaluate to an invalid address (including NULL) has type *pointer*. For example, the expression `&x + k` has type *pointer*.

***zero*:** An expression that is guaranteed to evaluate to 0 has type *zero*. For example, the literal `0` has type *zero*.

Nodes are connected by three kinds of (directed) assignment edges: *conversion edges* ($\xrightarrow{cvt}$), *extension edges* ($\xrightarrow{ext}$), and *copy edges* ($\xrightarrow{=}$). Conversion edges represent assignments with a right-hand side of the form $(\tau)_{cvt}e$, extension edges represent assignments with a right-hand side of the form $(\tau)_{ext}e$, and copy edges represent assignments that do not involve a type-cast, or that involve a type-cast of the form $(\tau)_{cpy}e$.

Figure 2(a) shows the mapping *AbsObj* from program expressions to abstract objects, while Figure 2(b) gives the rules for adding edges to the graph. For example, the assignment $x = y \oplus z$ adds to the graph two copy edges, $x \xleftarrow{=} \oplus_\tau y$ and $x \xleftarrow{=} \oplus_\tau z$, where τ is the static type of the expression $y \oplus z$.

Figure 3 shows the assignment graph that would be built for the example code given earlier in Figure 1.

3.2.2 Step 2: Computing runtime types

After building the assignment graph, the analysis computes a *runtime-type* attribute for each node in the graph. The values of *runtime-type* form the lattice shown in Figure 4. Intuitively, the *runtime-type* for node n summarizes the set of types that the expression corresponding to n might have at runtime. A *runtime-type* of $\perp$ means that an expression could have more than one

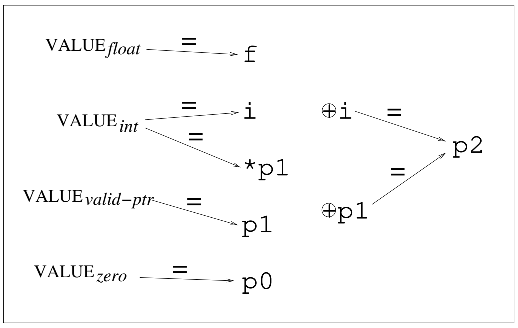


Fig. 3. The assignment graph for the example in Figure 1.

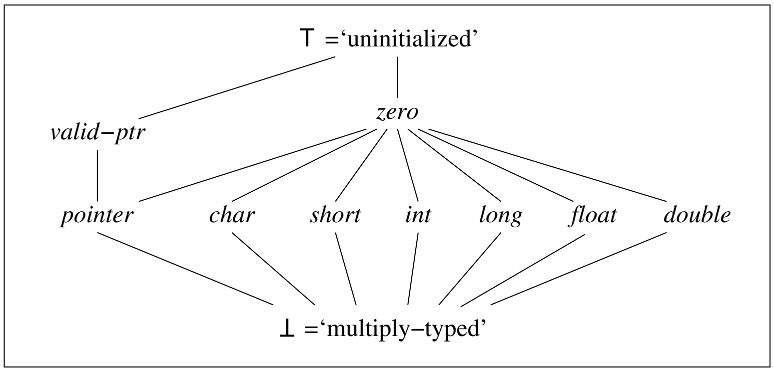


Fig. 4. The lattice for *runtime-type*.

incompatible runtime type.

Figure 5 gives the constraints for computing the *runtime-type* of each node in the assignment graph. In the figure, *pt-set*(p) is the points-to set of p, while *static-type*(n) is the static type of the expression represented by n.

Rules T1 and T2 set the *runtime-type* of each VALUE$_\tau$ node and each $\oplus_\tau x$ node to be its static type (τ). Rule T3 constrains the *runtime-type* of a $*p$ node to be no higher in the lattice than the type of each variable in p's points-to set. Rule T4 constrains the *runtime-type* of the left-hand side of a conversion to be no higher than its static type; this is safe because the RTC instrumentation will check the type of the right-hand side, and correct the runtime tag if there is an error. Rule T5 deals with extension edges: if the right-hand side of an extension assignment is well-typed, then the *runtime-type* of the left-hand side is constrained to be no higher than its static type; otherwise, the *runtime-type* of the left-hand side is set to $\perp$ (because such an assignment may potentially copy complicated RTC tags into the mirror of the left-hand side). Rules T6a and T6b handle assignment edges: if the left-hand side is a dereference of a pointer p (rule T6a), then the *runtime-type* of each node in the points-to

	Condition	Inferred Constraint
T1.		$\mathit{runtime\text{-}type}(\text{VALUE}_\tau) = \tau$
T2.		$\mathit{runtime\text{-}type}(\oplus_\tau x) = \tau$
T3.	$x \in \mathit{pt\text{-}set}(p)$	$\mathit{runtime\text{-}type}(*p) \sqsubseteq \mathit{runtime\text{-}type}(x)$
T4.	$n_1 \xleftarrow{cvt} n_2$	$\mathit{runtime\text{-}type}(n_1) \sqsubseteq \mathit{static\text{-}type}(n_1)$
T5.	$n_1 \xleftarrow{ext} n_2$	**if** $\mathit{runtime\text{-}type}(n_2) = \mathit{static\text{-}type}(n_2)$ **then** $\mathit{runtime\text{-}type}(n_1) \sqsubseteq \mathit{static\text{-}type}(n_1)$ **else** $\mathit{runtime\text{-}type}(n_1) = \bot$
T6a.	$*p \xleftarrow{=} n_2$, $x \in \mathit{pt\text{-}set}(p)$	$\mathit{runtime\text{-}type}(x) \sqsubseteq \mathit{runtime\text{-}type}(n_2)$
T6b.	$n_1 \xleftarrow{=} n_2$, n_1 not of the form $*p$	$\mathit{runtime\text{-}type}(n_1) \sqsubseteq \mathit{runtime\text{-}type}(n_2)$

Fig. 5. Rules for computing *runtime-type*.

Assignments	Assignment Edges	Inferred Constraints
`i = 1;`	$i \xleftarrow{=} \text{VALUE}_{int}$	$\mathit{runtime\text{-}type}(i) \sqsubseteq \mathit{int}$
`f = 2.3;`	$f \xleftarrow{=} \text{VALUE}_{float}$	$\mathit{runtime\text{-}type}(f) \sqsubseteq \mathit{float}$
`p0 = 0;`	$p0 \xleftarrow{=} \text{VALUE}_{zero}$	$\mathit{runtime\text{-}type}(p0) \sqsubseteq \mathit{zero}$
`p1 = &i;`	$p1 \xleftarrow{=} \text{VALUE}_{valid-ptr}$	$\mathit{runtime\text{-}type}(p1) \sqsubseteq \mathit{valid\text{-}ptr}$
`p1 = (int *) &f;`	$p1 \xleftarrow{=} \text{VALUE}_{valid-ptr}$	$\mathit{runtime\text{-}type}(p1) \sqsubseteq \mathit{valid\text{-}ptr}$
`p2 = p1 + i;`	$p2 \xleftarrow{=} \oplus_{pointer}\ p1$ $p2 \xleftarrow{=} \oplus_{pointer}\ i$	$\mathit{runtime\text{-}type}(p2) \sqsubseteq \mathit{pointer}$ $\mathit{runtime\text{-}type}(p2) \sqsubseteq \mathit{pointer}$
`*p1 = 4;`	$*p1 \xleftarrow{=} \text{VALUE}_{int}$ $(\mathit{pt\text{-}set}(p1) = \{i, f\})$	$\mathit{runtime\text{-}type}(*p1) \sqsubseteq \mathit{runtime\text{-}type}(i)$ $\mathit{runtime\text{-}type}(*p1) \sqsubseteq \mathit{runtime\text{-}type}(f)$ $\mathit{runtime\text{-}type}(i) \sqsubseteq \mathit{int}$ $\mathit{runtime\text{-}type}(f) \sqsubseteq \mathit{int}$

Final *runtime-types*:	
$\mathit{runtime\text{-}type}(p0) = \mathit{zero}$	$\mathit{runtime\text{-}type}(i) = \mathit{int}$
$\mathit{runtime\text{-}type}(p1) = \mathit{valid\text{-}ptr}$	$\mathit{runtime\text{-}type}(f) = \bot$
$\mathit{runtime\text{-}type}(p2) = \mathit{pointer}$	$\mathit{runtime\text{-}type}(*p1) = \bot$

Fig. 6. Computing *runtime-types* for the example in Figure 1.

set of p can be no higher than the *runtime-type* of the right-hand side; if the left-hand side is a variable v, then its *runtime-type* can be no higher than the *runtime-type* of the right-hand side.

Figure 6 illustrates the computation of *runtime-types* for the example of Figure 1.

	Condition	Attribute
L1.	*runtime-type*$(n) \not\supseteq$ *static-type*(n)	n *unsafe*
L2.	*static-type*(p) = *pointer*, *runtime-type*$(p) \neq$ *valid-ptr*	$*p$ *unsafe*
L3.	$*p$ *unsafe*, $x \in$ *pt-set*(p), $x \neq$ *unsafe*	x *tracked*

Fig. 7. Rules for determining *type-safety* attributes.

3.2.3 Step 3: Computing type-safety levels

Once the *runtime-types* are computed, each node of the graph is annotated with an attribute signifying its type-safety level — either *unsafe* or *tracked* — based on the rules given in Figure 7; after applying the rules, any node not marked either *unsafe* or *tracked* is considered *safe*.

Rule L1 marks *unsafe* any node whose *runtime-type* is not compatible with its *static-type*. For Rule L2, a pointer p whose *runtime-type* is not *valid-ptr* may contain an invalid address; therefore $*p$ must be instrumented to check its runtime type, and is thus marked *unsafe*. Rule L3 marks *tracked* any variable in the points-to set of a pointer p whose dereference node ($*p$) is *unsafe*.

Looking back at the example in Figure 6, Rule L1 makes f and $*p1$ *unsafe*, L2 makes $*p0$ and $*p2$ *unsafe*, and L3 makes i *tracked*. This leaves $p0$, $p1$ and $p2$ as *safe*.

It is expected that a significant proportion of the expressions in a program will be *safe*, and that the elision of type-checking instrumentation for those expressions will result in a significant performance improvement in the RTC tool.

Note that it is possible for the contents of a *safe* variable v to be accessed indirectly by an errant pointer p. However, in such a case, $*p$ will have been marked *unsafe*, and thus will be instrumented with a *verify-pointer* operation. Since *safe* variables are not instrumented, v's mirror will be tagged *unallocated*, so the check of $*p$ will trigger an "accessing unallocated memory" error. Since more memory will be tagged as *unallocated* "red zones" than before, this could potentially catch more errors than the unoptimized RTC tool.

3.3 Flow-Sensitive Refinements

This section describes three flow-sensitive dataflow analyses that complement the flow-insensitive type-safety-level analysis defined above. In the first case, may-be-uninitialized analysis is needed to ensure the soundness of the instrumentation elimination proposed above. In the second case, never-null-dereference analysis allows the above analysis to classify more expressions of the form $*p$ as *safe*, thus allowing more instrumentation to be eliminated. In the third case, redundant-checks analysis discovers opportunities for further elimination of runtime checks, in a direction orthogonal to the above approaches. All three analyses can be described in terms of a standard dataflow analysis; they are independent, and could be performed in parallel.

3.3.1 May-be-uninitialized analysis

The type-safety-level analysis described in Section 3.2 does not account for uses of uninitialized data, which is an error currently detected by the RTC tool. That is, by eliminating all instrumentation for `safe` and `tracked` locations, and by initializing `tracked` locations to their static types rather than to *uninitialized*, the RTC tool will no longer detect uses of uninitialized data in these locations. To address this problem, an additional flow-sensitive analysis is needed to find program points where instrumentation cannot be elided.

For a location x that is *safe* or *tracked*, the analysis finds instances of x where x may be uninitialized. This analysis uses dataflow facts of the form *uninit*(x), which means x may be uninitialized. The fact *uninit*(x) is generated for a local variable x at the program point where x is declared. An assignment $x = y$ that is reached by an *uninit*(y) fact generates an *uninit*(x) fact; otherwise, an assignment into x kills *uninit*(x). For an assignment via a pointer, we use the points-to sets already computed, and err on the side of safety: if the assignment $*p = y$ is reached by *uninit*(y), we generate *uninit*(v) for each v in p's points-to set; if not, we do not kill any *uninit*(v) facts. The meet operator is set union.

A `tracked` or `safe` location x for which some use of x is reached by an *uninit*(x) fact needs to be treated specially:

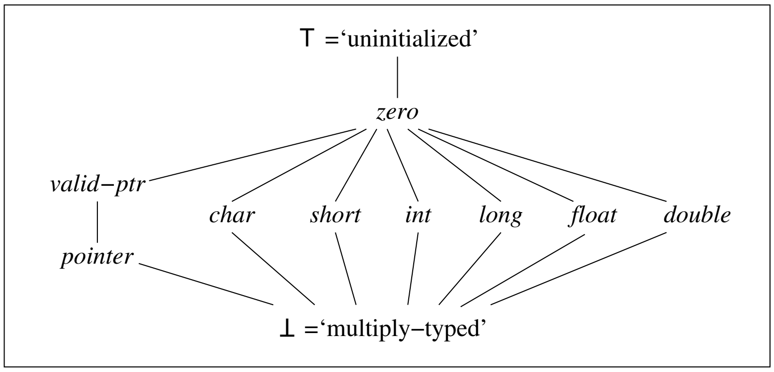
- The declaration of x is instrumented with a *declare* operation that sets its type in the mirror to *uninitialized*.
- Every use of x that is reached by an *uninit*(x) fact is instrumented with a *verify* operation.
- Every definition of x that might change x's status (from uninitialized to initialized, or vice versa) is instrumented with a *copy* operation (this ensures that x's tag is set correctly for subsequent uses of x). The only definitions that are guaranteed *not* to change x's status are those for which *uninit*(x) holds neither before nor after the definition, thus, all other definitions of x are instrumented.

3.3.2 Never-null-dereference analysis

Another shortcoming of the approach of Section 3.2 is that if a pointer p is ever assigned a NULL value, then $*p$ is marked *unsafe* (and must thus be fully instrumented). This is because the NULL constant maps to the VALUE_{zero} object, whose *runtime-type* is *zero*. The assignment to `p` causes `p`'s *runtime-type* to be constrained to be no higher in the lattice than *zero*. This prevents `p`'s *runtime-type* from being *valid-ptr*; thus, by Rule L2 in Figure 7, `*p` is marked *unsafe*. This does not cause the RTC tool to miss any errors or to report any spurious errors, but it adds to the runtime overhead by including some unnecessary instrumentation.

With another flow-sensitive refinement, we can find some instances of $*p$ where p is guaranteed not to contain a NULL value, and eliminate instrumentation for checking those instances of $*p$.

First, the *runtime-type* lattice (in Figure 4) is modified so that the *valid-ptr* type is below the *zero* type, as follows:



Second, Rule L2 in Figure 7 is changed to:

	Condition	Attribute
L2.	*static-type*(p) = *pointer*, *runtime-type*(p) $\not\sqsupseteq$ *valid-ptr*	$*p$ *unsafe*

With these modifications, for any pointer p that is only assigned valid pointer values or a NULL value, $*p$ will be *safe*. (Note that if a pointer may be assigned the result of pointer arithmetic or other invalid pointer values, then $*p$ will still be considered *unsafe*.)

Next, we perform another dataflow analysis to account for possible null-pointer dereferences of these pointers. The dataflow fact *null*(p) is generated for each assignment of a NULL value into a pointer p. An assignment of the form p = `&`x kills *null*(p). For an assignment $q = p$ that is reached by *null*(p), *null*(q) is generated. For assignments via pointers, we make safe approximations as in the may-be-uninitialized analysis. The transfer function for a predicate with a null-comparison condition (e.g., `p == 0`) propagates a *null*(p) fact along one branch, and kills it on the other, as appropriate. Again, the meet operator is set union.

Upon completion of the analysis, a *safe* dereference $*p$ that is reached by a *null*(p) must be instrumented to perform a null-pointer check.

3.3.3 Redundant-checks analysis

One further flow-sensitive refinement is to eliminate *redundant* checks. When a variable v is read many times with no intervening writes, a runtime check is only necessary for the first read of v. Such situations can be detected with the following dataflow analysis: a CFG node n that is instrumented to check variable v for type τ generates a dataflow fact $\mathit{check}(v, \tau)$, while a node containing an unsafe assignment into v (where the right-hand-side is *unsafe*) kills all $\mathit{check}(v, \tau)$ facts. The analysis must again be safe for assignments via pointers: for an assignment into $*p$, if v is in p's points-to set, then any $\mathit{check}(v, \tau)$ fact is killed. After the analysis, if the dataflow fact $\mathit{check}(v, \tau)$ reaches another node n_1 that is to be instrumented to check v for type τ, the check at n_1 may be eliminated. This is only safe if a reaching fact $\mathit{check}(v, \tau)$ means that the node is *always* reachable by another check of v for type τ. Therefore, the *meet* operator needs to be set intersection.

4 Evaluating Potential for Optimizations

To estimate the potential speedup that could be gained from the type-safety-level analysis, we implemented a simplified version that classifies type-safety levels as follows: first, every variable whose address is taken and every pointer dereference is marked *unsafe*; then the *unsafe* annotations are propagated along assignment edges (that is, if the edge $n_1 \xleftarrow{=} n_2$ is in the assignment graph, and n_2 has been marked *unsafe*, then n_1 is also marked *unsafe*). Upon completion of the analysis, expressions that map to abstract objects not marked *unsafe* are considered *safe*, and are not instrumented with RTC checks. This optimization gives a lower bound on the amount of instrumentation that could be eliminated by the full type-safety-level analysis.

Figure 8 presents execution times for several benchmarks used in [9], and several SPEC benchmarks, averaged over three runs on different test inputs. Running times are given for the uninstrumented executables (column (a)), for the unoptimized RTC-instrumented executables (column (b)), and for the RTC-instrumented executables optimized using the results of the simple analysis (column (c)). Columns (d) and (e) give the slowdown factors for the RTC-instrumented executables relative to the uninstrumented executable (e.g., for `aes`, the unoptimized RTC-instrumented version runs 21.63 times slower than the uninstrumented executable, while the optimized RTC-instrumented version runs 11.46 times slower). Column (f) gives the percentage speedup of the optimized RTC executable over the unoptimized one.

On average, the unoptimized RTC executables ran with a slowdown factor of 37.4 times the execution time of the uninstrumented executables. The optimized RTC executables ran with a slowdown of only 25.1 times, corresponding to an average speedup of 39.9% over the unoptimized RTC executables. This result shows that even with a simple conservative analysis we can achieve significant speedup.

Figure 9 presents some details about the results of the simplified analysis. Column (a) gives the number of abstract objects in each benchmark, and column (b) gives the number of those objects marked *unsafe* by the analysis. Overall, 41.0% of the abstract objects are marked *unsafe*. On the one hand, this number indicates that even the simplified analysis was able to classify

Benchmark	Running Times (ms)			Slowdown Factor		(f)
	(a) No RTC	(b) Unopt RTC	(c) Opt RTC	(d) Unopt RTC	(e) Opt RTC	Opt % Speed-up
aes	5,195	112,378	59,545	21.63	11.46	47.01
cacm	8,319	194,069	68,093	23.33	8.18	64.91
cfrac	8,138	438,254	301,391	53.85	37.03	31.23
matxmult	4.610	32,375	17,159	7.02	3.72	47.00
ppm	5,895	210,689	184,582	35.74	31.31	12.39
compress	31,467	976,861	344,684	31.04	10.95	64.72
go	14,805	793,880	587,306	53.62	39.67	26.02
ijpeg	2,102	76,761	32,620	36.52	15.52	57.50
li	4,599	339,382	312,817	73.79	68.01	7.83

Fig. 8. Comparison of running times for the uninstrumented executable, the unoptimized RTC-instrumented executable, and the optimized RTC-instrumented executable. The slowdown factor is in comparison to the uninstrumented executable.

Benchmark	(a) # abs objs	(b) # unsafe objs	(c) # deref objs	(d) % singl pt-sets
aes	705	214	142	73.65
cacm (decode)	120	30	18	50.00
cfrac	1781	657	209	52.56
matxmult	122	29	24	25.93
ppm (encode)	881	374	65	86.53
compress	557	152	59	62.12
go	15096	5783	2939	22.61*
ijpeg	14130	5564	2217	15.11*
li	5406	3087	383	3.44*
				*numbers from [5].

Fig. 9. Results of the simplified analysis.

the majority of the abstract objects as *safe*. On the other hand, there are still many objects marked *unsafe* that might potentially be marked *safe* by the full type-safety-level analysis. To estimate this potential, we considered how using a points-to analysis such as the one defined by Andersen [1] might improve our results.

Note that our simplified analysis very closely approximates the results of performing the full type-safety-level analysis using a degenerate points-to analysis, in which every object whose address is taken is in the points-to set of every object that is ever dereferenced.[4] Column (c) of Figure 9 gives the

[4] The two analyses differ only if all address-taken variables are of the same type, in which case the full analysis with the degenerate points-to set would classify those variables as *safe*,

number of dereference objects in each benchmark (abstract objects of the form $*e$, all of which are marked *unsafe* by the simplified analysis). On average, those objects accounted for 15.6% of all abstract objects. We conjectured that a non-trivial points-to analysis would give well-typed points-to sets for a large proportion of those dereference objects, hence causing them to be marked *safe*, and transitively resulting in a greater number of other abstract objects being marked *safe*.

To evaluate this conjecture, we performed Andersen's points-to analysis, as implemented in [21], on the benchmarks, and measured the percentage of dereference objects whose points-to sets contain exactly one element. This is given in column (d) of Figure 9 (the numbers for the last three benchmarks were obtained from [5]; due to limitations in our implementation we were unable to analyze those programs). On average, 43.6% of the dereference objects have singleton points-to sets. When a pointer's points-to set contains a single element, both the pointer and the pointer target are most likely well-typed, and the two will therefore likely be *safe* locations. Therefore, this column suggests the minimum percentage of potential reduction in the number of dereference objects marked *unsafe* when we replace our simplified analysis with the full type-safety-level analysis. For example, for `aes`, 73.65% of the dereference objects had singleton points-to sets; therefore, we would expect at least 73.65% of the 142 dereference objects in the program (all marked *unsafe* by the simplified analysis) to be marked *safe* by the full type-safety-level analysis, thus decreasing the total number of *unsafe* objects from 214 to 109. Note that this estimate is overly conservative: even if a pointer's points-to set contains more than one object, if all of the objects are well-typed, then they will all be *safe*, and even more of the objects marked *unsafe* by the simplified analysis will be marked *safe* by the full type-safety-level analysis.

5 Related Work

Many runtime approaches have been proposed and developed that instrument a program to track auxiliary information during program execution.

Purify [7] and Insure++ [14] are two commercial products that have proven to be successful in detecting many classes of memory errors at runtime. Those tools can cause a runtime slowdown of 20 times or more; it is possible that using techniques similar to those presented here could decrease that slowdown.

Austin *et al.* [2] describe the *Safe C* system which tracks information about each pointer's referent, and uses this information to detect spatial (e.g., array out-of-bounds) and temporal (e.g., stale pointer dereference) access errors. Patil *et al.* [15] propose a novel way to make this check more efficient by performing the tracking and checking of the auxiliary information in a *shadow process* on a separate processor.

while the simplified analysis would not.

Cyclone [9] and *CCured* [13] are two systems based on the C language that attempt to inject some level of safety while maintaining the low-level control of the language. The *Cyclone* language includes the definition of different kinds of pointers with different safety restrictions; “unsafe” pointer dereferences are instrumented with runtime checks (using “fat pointers” in a manner similar to *Safe-C*). To port an existing C program into *Cyclone*, the programmer must manually convert C pointers into the appropriate kind of Cyclone pointer to achieve optimal performance; an analysis to automatically classify pointers into the appropriate safety level, similar to the type-safety-level analysis proposed in this paper, would make it easier to port existing C code, and thus encourage greater use of this new language.

CCured also includes runtime checks for bad pointer dereferences. *CCured*'s checks are more limited than RTC checks: specifically, *CCured* focuses only on pointers, and does not differentiate non-pointer types. Furthermore, *CCured* can be too strict (i.e., certain valid program behavior, such as storing the address of a stack variable in a global variable, or storing a pointer value in an integer, casting it back, and dereferencing it, will cause a runtime check to fail).

To reduce the overhead of runtime checks, *CCured* uses a type-inference scheme to identify as many *safe* and *sequence* pointers as possible, thus minimizing the amount of instrumented operations. The goal of their type inference is thus similar to that of our type-safety-level analysis. However, their type-inference scheme is less precise than our proposed analysis: they effectively group points-to sets into equivalence classes (in the spirit of Steensgaard's points-to analysis [18]), while our analysis accounts for the directionality of assignments. Despite this, they were able to significantly improve the performance of instrumented *CCured* programs, from the unoptimized slowdown of 6-20 times, to between 1 and 2 times slowdown using the type-inference optimization. This suggests the potential of our analysis achieving comparable or better performance improvements.

The use of runtime checks to enforce safety properties, and techniques for eliminating unnecessary checks to improve performance, have been used in other programming languages and environments. Implementations of dynamically-typed languages like LISP and Scheme need to maintain runtime information to perform runtime type-checking as part of the language's semantics. To improve the performance of such a system, Henglein [8] proposes an efficient approach based on type inference. The Java language needs to perform potentially expensive runtime checks, such as array-bounds checks, to enforce safety properties guaranteed by the language. The elimination of redundant and unnecessary array-bounds checks in Java and other safe languages has been studied extensively [19,12,6,3,4,11].

6 Conclusions

We have presented some techniques for reducing the runtime overhead of the instrumentation added to C programs by the Runtime Type-Checking tool. In Section 3.2, we defined a flow-insensitive type-safety-level analysis, which marks each expression in the program *safe*, *unsafe*, or *tracked*. The results of that analysis, used in conjunction with the results of the may-be-uninitialized analysis defined in Section 3.3.1, can be used to remove most of the instrumentation for *safe* and *tracked* expressions.

Two further flow-sensitive analyses were presented in Sections 3.3.2 and 3.3.3 to eliminate more instrumentation. The first, never-null-dereference analysis, eliminates checks for pointers that can be statically determined never to be null. The second identifies redundant checks that can be eliminated as well.

To gauge the potential benefits to be gained by these optimizations, we implemented a simplified version of the type-safety-level analysis, and presented experimental results that suggest there is much potential for improving the performance of the RTC tool. It is possible that similar ideas can be adapted for other tools that involve detecting errors via runtime instrumentation.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[2] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '94), SIGPLAN Notices 29(6)*, pages 290–201, Orlando, FL, June 1994.

[3] R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00), SIGPLAN Notices 35(5)*, pages 321–333, Vancouver, BC, June 2000.

[4] Wei-Ngan Chin, Siau-Cheng Khoo, and Dana N. Xu. Deriving pre-conditions for array bound check elimination. In *Proceedings of the Second Symposium on Programs as Data Objects, PADO 2001*, pages 2–24, Aarhus, Denmark, May 2001.

[5] Manuvir Das. Unification-based pointer analysis with directional assignments. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00), SIGPLAN Notices 35(5)*, pages 35–46, Vancouver, BC, June 2000.

[6] Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, March–December 1993.

[7] R. Hasting and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter Usenix Conference*, 1992.

[8] Fritz Henglein. Global tagging optimization by type inference. In *LISP and Functional Programming*, pages 205–215, 1992.

[9] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[10] A. Loginov, S. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. In *Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *Lec. Notes in Comp. Sci.*, pages 217–232. Springer, April 2001.

[11] Mikel Lujan, John R. Gurd, T. L. Freeman, and Jose Miguel. Elimination of java array bounds checks in the presence of indirection. Technical Report CSPP-13, Department of Computer Science, University of Manchester, February 2002.

[12] Victoria Markstein, John Cocke, and Peter Markstein. Optimization of range checking. In *ACM SIGPLAN Symposium on Compiler Construction, SIGPLAN Notices 17(6)*, pages 114–119, Boston, MA, June 1982.

[13] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *ACM Symposium on Principles of Programming Languages*, Portland, OR, January 2002.

[14] Parasoft. Insure++: An automatic runtime error detection tool, 2001. http://www.parasoft.com/ insure/papers/tech.htm.

[15] H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software–Practice and Experience*, 27(27):87–110, 1997.

[16] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *Proc. of ESEC/FSE '99: Seventh European Softw. Eng. Conf. and Seventh ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, pages 180–198, September 1999.

[17] R. Stallman and R. Pesch. *Using GDB: A Guide to the GNU Source-Level Debugger*. July 1991.

[18] Bjarne Steensgaard. Points-to analysis in almost linear time. In *ACM Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[19] Norihisa Suzuki and Kiyoshi Ishihata. Implementation of an array bound checker. In *ACM Symposium on Principles of Programming Languages*, pages 132–143, Los Angeles, CA, January 1977.

[20] D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Symposium on Network and Distributed Systems Security (NDSS '00)*, pages 3–17, San Diego, CA, February 2000.

[21] S. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '90), SIGPLAN Notices 25(6)*, pages 91–103, Atlanta, GA, May 1999.