Run-Time Type Checking for Binary Programs*

Michael Burrows[1], Stephen N. Freund[2], and Janet L. Wiener[3]

[1] Microsoft Corporation, 1065 La Avenida, Mountain View, CA 94043
[2] Department of Computer Science, Williams College, Williamstown, MA 01267
[3] Hewlett-Packard Labs, 1501 Page Mill Road, Palo Alto, CA 94304

Abstract. Many important software systems are written in the C programming language. Unfortunately, the C language does not provide strong safety guarantees, and many common programming mistakes introduce type errors that are not caught by the compiler. These errors only manifest themselves at run time through unexpected program behavior, and it is often hard to isolate and identify their causes. This paper presents the Hobbes run-time type checker for compiled C programs. Our tool interprets compiled binaries, tracks type information for all memory and register locations, and reports warnings when a variety of type errors occur. Because the Hobbes type checker does not rely on source code, it is effective in many situations where similar tools are not, such as when full source code is not available or when C source is linked with program fragments written in assembly or other languages.

1 Introduction

Many software systems are written in the C programming language because it is expressive and provides precise, low-level control over the machine architecture. However, this strength is also a weakness. The expressive power of C is obtained through unsafe language features, including pointer arithmetic, explicit memory management, unchecked type casts, and so on. These features give the programmer a great deal of control but also make it difficult to ensure software reliability and to maintain large programs.

Given the importance of many systems in this category, it is essential to identify defects caused by improper use of unsafe language features. In this paper, we present Hobbes, a new run-time analysis tool that identifies a large class of errors in compiled C programs. In particular, our tool identifies *memory access errors* and *type errors*. A memory access error occurs when a program accesses an invalid memory location. Two examples of such errors are (1) reading from or writing to an unallocated location, and (2) reading from an allocated but uninitialized location. A type error occurs when an operation is performed on operands whose types are incompatible with the operation. Adding a pointer to a real number, calling a function with the wrong number or type of arguments, and dereferencing an integer as a pointer are all type errors.

* This work was performed, in part, while all 3 authors were employed at the Compaq Systems Research Center (now part of HP Labs).

G. Hedin (Ed.): CC 2003, LNCS 2622, pp. 90–105, 2003.

To catch errors, our tool maintains a shadow memory containing the allocation status and type of each location accessible to the target program, which it updates and checks as the target is running. Purify demonstrated the effectiveness of a shadow memory-based approach for identifying memory access errors [3]. Purify modifies the target program to maintain allocation and initialization status for each memory location, and it instruments each memory operation to check that the status information for the address being accessed is in an appropriate state. The Hobbes type checker goes beyond Purify by tracking not only memory status information, but also the type stored at each location. The type information enables our tool to check the types of the operands for each operation performed as the program executes.

The Hobbes prototype checks for errors in Linux binaries on the Intel x86 architecture. Hobbes consists of two major components: an instrumentable x86 interpreter and a run-time type checker. To check a program for type errors, the type checker maintains the shadow memory and checks each interpreted instruction for errors. Memory access and type errors are reported to the programmer, along with the call stack and the relevant data values and types.

The type checker extracts type information from the symbol tables and debug tables embedded in the binary program. It uses this information to determine the types of storage locations allocated to global variables, local variables, and parameters of functions. When debugging information is incomplete or not available, the type checker assumes more conservative types for memory locations. Even when given only partial type information for the target program, Hobbes can still identify a useful set of errors.

The Hobbes architecture provides the following benefits:

1. Hobbes uses only the binary representation of programs and does not rely on the source code for the target program or included libraries.
2. Hobbes is applicable to programs written in a mixture of any languages that compile into the standard binary format.
3. Hobbes does not modify the data representations or layout of the program.

We are not aware of other tools that provide all three of these benefits. Loginov et al. present a system similar to ours that employs source-to-source translation to insert code to maintain and check shadow memory [7]. Relying on source code translation limits their handling of libraries and mixed-language programs, and their tool does not preserve the instruction stream of the original program. Several other tools have been proposed to check for memory access errors and some type errors by extending the representation of pointers to include additional information (see, for example, [15,1]). However, we wished to avoid changing the data layout of the program since such changes are not always feasible in large systems.

Our experience indicates that the Hobbes type checker is an effective tool for finding type errors in programs. When applied to a set of programs from an undergraduate compilers class, it found a number of both memory errors and type errors, and it scaled reasonably well when checking larger programs. In particular, the false alarm rate was not a significant impediment to using the tool.

There is a substantial performance penalty for using the Hobbes type checker prototype, but we are confident that improvements we describe will significantly improve the performance of the system.

Section 2 motivates this work by demonstrating how run-time type checking can catch a number of common errors in C programs. Sections 3 and 4 describe the general Hobbes architecture and the type checker, respectively. We summarize our experiments to validate the type checker and measure performance in Section 5, and Section 6 compares the Hobbes type checker to related work. We conclude in Section 7 and outline directions for future work.

2 Motivating Examples

In this section, we present some errors that the Hobbes type checker catches, but which are not caught by the C compiler's static type checking or the allocation checking performed by tools like Purify. Figure 1 contains programs exhibiting these errors. In each case, we outline how the errors are caught.

In Example 1, the programmer writes a pointer into a union but then reads the union value as an integer. On the store to `x.p`, the type checker sets the shadow memory for that location to `pointer`. The type `pointer` is inferred because a `lea` (load effective address) instruction is used to compute `&i`. A multiply instruction can not be applied to an operand of type `pointer`, so when the multiply of `x.k` occurs, the type checker detects the type mismatch and generates a warning message. This example is interesting because it shows that useful type checking can be done without any help from the compiler or debugging information.

Example 2 shows an array bounds error that is not normally detected by Purify or similar systems. The programmer writes to `y.a[10]`, which is beyond the end of the array, but still part of an allocated structure. The assignment overwrites the field `y.h`, which follows the array in memory. If debugging information is included in the program binary, the type checker knows that `y.h` should have type `int`. It reports an error when the program writes a value of type `pointer` instead. If no debugging information is available, the write is permitted, but the type checker detects an error when the value of type `pointer` in `y.h` is later used in a multiplication.

Example 3 shows a common pitfall in the use of the standard C library sorting function, `qsort()`. The comparison function required by `qsort()` is called with pointers to the elements to be compared, rather than the elements themselves. The naive programmer who wrote Example 3 omitted this extra level of indirection. A cast is almost always required when using `qsort()`, and the one used here, though not unusual, masks the error. Given the debugging information for the program, the type checker expects values of type `int` for each parameter of `cmpint()`. When values of type `pointer` are passed instead, it generates a warning.

Example 1:

```
union {
  int k;
  int *p;
} x;

void ex1() {
  int i, j;
  x.p = &i;
  j = 17 * x.k;
}
```

Example 2:

```
struct {
  int *a[10];
  int h;
} y;

void ex2() {
  int i, j;
  for (i = 0; i <= 10; i++)
    y.a[i] = &j;
  y.h *= 10;
}
```

Example 3:

```
int cmpint (int a, int b) { return ((b < a) - (a < b)); }

void ex3()
  int i;
  int array[N];
  ...
  qsort (array, N, sizeof (array[0]),
     (int (*) (const void *, const void *)) cmpint);
}
```

Fig. 1. C programs with type errors.

3 The Hobbes System Architecture

Hobbes consists of two distinct pieces: an x86 *interpreter* that runs the target program and the *type checker* analysis tool—a module that is called by the interpreter when events of interest occur in the target. The operating system kernel, in this case Linux, is unmodified. In this section, we describe the interpreter. In the next section, we describe the type checker.

The Hobbes platform is a general framework in which to build analysis tools like the Hobbes type checker. The interpreter plays the same role as a binary editor like Atom [14], or the instrumentable dynamic compiler that underlies Valgrind [12]. An analysis tool first registers interest in events that may occur while the target is running. For example, a tool may indicate that it wants notification each time the target accesses memory or executes a specific opcode. The interpreter then runs the target, which is unaware that instrumentation is taking place, and calls analysis routines provided by the tool when interesting events occur. Arguments to the analysis routines convey relevant information about the event, indicating any memory addresses, values, and registers involved.

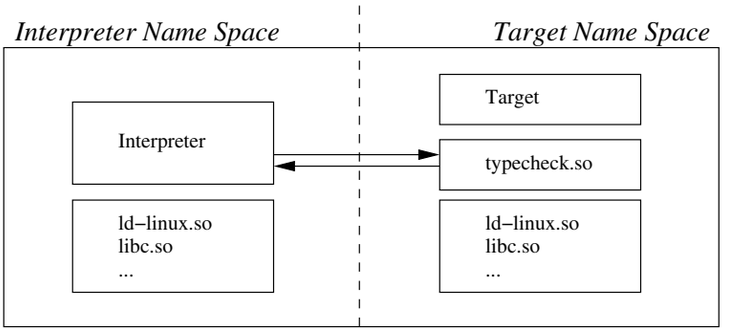


Fig. 2. The Hobbes system architecture.

A major goal of Hobbes is to provide a program environment for the target that is as close as possible to its normal execution environment. This goal influenced our design in two ways. First, we placed the components of Hobbes in normally unused parts of the address space to avoid having to relocate the target. Second, Hobbes uses two distinct name spaces, as illustrated in Figure 2. The target name space contains the target program, the type checker tool (which is a shared library) and other libraries required by the target. The interpreter name space contains the interpreter and the (potentially different) set of libraries that are linked with it. This separation prevents problems arising from name clashes or version mismatches between the libraries used in the interpreter and those used in the target, as well as potential interference problems caused by the interpreter and target sharing library data structures. Analysis tools reside in the target name space to give them access to the target dynamic loader, which is used to resolve target addresses to names.

The interpreter name space is created by the Linux kernel. To create the target name space, the Hobbes interpreter simulates the actions that the kernel would have taken to run the target, including running a new copy of the dynamic loader (`ld-linux.so`). This second loader loads the target, its shared libraries, and the analysis tools.

The Hobbes interpreter is written entirely in x86 assembly language. The main loop in the interpreter fetches instructions from the target instruction stream and performs computed jumps into tables whose entries are code fragments. The code fragment for each instruction:

1. decodes the operand specifiers and loads the addresses of the operands into specific registers,
2. performs the operation defined by the instruction opcode on those registers, and
3. moves the result, if the instruction has one, to its ultimate destination.

Except in rare circumstances, the core of the implementation for each opcode is performed by the corresponding x86 instruction. This method allows side-effects,

such as the setting of condition codes, to be captured faithfully, and it improves the chances of correctly emulating the execution of unusual code sequences or instructions that behave differently on different x86 implementations.

The shared libraries for tools register analysis routines with the interpreter when the libraries are initialized as part of the loading process. If a tool has registered an analysis routine for a particular instruction opcode, a call to the analysis routine is inserted into the appropriate table entry between the first and second step above. The analysis routines are executed directly, not interpreted. The interpreter does not interpret operating system kernel code. When the interpreter encounters a system call, it executes a kernel trap in the normal way, after loading the registers with the arguments needed for the call. Some calls, notably those dealing with signals, are handled specially so as to maintain control of the target program.

Other analysis frameworks, such as Atom and Valgrind, employ binary code modification techniques to avoid the overhead of interpreting machine code. Although we could have adopted these techniques to obtain better performance, we chose to implement the interpreter for several reasons. First, interpretation preserves the layout and location of the target code and data segments, which reduces the likelihood of introducing unintended errors into the target during instrumentation. Also, no publicly available binary editor or dynamic compiler existed for x86 Linux when we started (Valgrind had not yet been released), and writing the interpreter was the simplest and fastest way to build a working prototype. In addition, the Hobbes type checker imposes a large overhead on execution beyond the interpreter's overhead, making the argument for more efficient instrumentation techniques less compelling. The large overhead is partially due to the type checker instrumenting virtually every instruction in order to track values as they pass through registers. In contrast, Purify only instruments memory accesses.

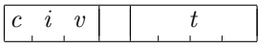
4 The Hobbes Type Checker

During startup, the Hobbes type checker shared library initializes its internal data structures and shadow memory and registers analysis routines with the interpreter. When an instruction of the target program is interpreted, the type checker tests the types of the operands and updates the type information in the shadow memory according to the instruction semantics. Any inconsistencies are reported to the user. In this section, we describe the shadow memory layout and data structures used by the type checker, demonstrate the steps to type check instructions and function calls, and describe features that reduce occurrences of false alarms. A false alarm occurs when the type checker incorrectly reports that a type error has occurred.

Shadow Memory and Type Representation. The x86 architecture provides a 4 GB address space, which the type checker divides into three sections. It uses addresses `0x00000000` – `0x5fffffff` for the target program memory and

addresses `0x60000000` – `0xbfffffff` for the shadow memory. The Linux kernel utilizes parts of the remaining 1 GB, and we do not use it in Hobbes. Each byte in the target program is matched by a byte in the shadow memory, which encodes its type. To map from a data address to its shadow address, the type checker simply adds `0x60000000` to the data address. The interpreter places the virtual registers in the target address section so that the interpreter can shadow them like all other locations accessible to the target.

The type checker tracks primitive C types. It currently represents structures, unions, and arrays as sequences of these primitive types and does not distinguish between different pointer types. Each primitive type is encoded in the shadow memory with a bit pattern equal to the size of the type in length. For example, the four-byte integer encoding covers four bytes of shadow memory. Each byte in the shadow memory contains four fields:

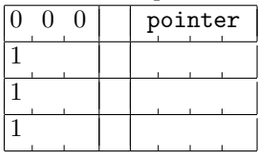


The *continuation bit* c is zero if the byte is the first byte of a type encoding, and one otherwise. When c is set, the other seven bits are unused. The *initialized bit* i indicates whether the corresponding data object has been initialized. The *invariant bit* v indicates whether the type of the corresponding data object may change during execution (i.e., whether the type encoding may be overwritten with a different type encoding). The type checker marks global data and stack locations for parameters and local variables as invariant when the debugging information supports doing so. The *base type* t encodes the type of the data. The type checker currently supports the primitive types `int8` (char), `uint8` (unsigned char), `unk8` (one byte of unknown type), `int16`, `uint16`, `unk16`, `int32`, `uint32`, `unk32`, `float`, `double`, and `pointer`. The `unallocated` base type indicates one byte of unallocated memory, and the `code` base type indicates one byte of code.

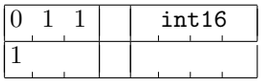
We present several type encodings to illustrate the layout of these structures:

initialized, unsigned char

uninitialized pointer



initialized, invariant short



Currently, there is unused space in the encodings for multi-byte types. However, these encodings enable a fast mapping function from each data value to its shadow memory and are easy to decode. In addition, we plan to use the remaining space to encode aggregate type and pointer information in the future.

Type Checker Initialization. The type checker performs the following initialization steps before the target program begins execution.

1. The type checker reads all available debugging information in the target and shared library object code. This information includes type declarations, function prototypes, global and local variable declarations, and the mapping from names to addresses.

2. For each type, the type checker creates a template block of shadow memory that encodes it, as outlined above. Hobbes creates the template block for aggregate types by concatenating the template blocks for each element. If part of a type is unknown or ambiguous, such as when a union may contain two different primitive types, the corresponding part of the template block contains the unknown type encoding of the appropriate size.
3. For each function, the type checker creates two template blocks, one for its parameters and one for its local variables. These blocks contain the shadow memory encodings for the function's activation record. Local variable locations that may contain different types at different points in the function are assigned the unknown type. All encodings in these blocks are invariant, except those for unknown types, which are not marked as invariant.
4. The type checker initializes the shadow memory for global variables with the template blocks created in step 2. All global variables are invariant, except those which have unknown type.
5. The type checker registers analysis routines for opcodes and system calls with the interpreter.

The type checker precomputes the type representations and template blocks to avoid translating types into their encodings at run time. The types for locations originally marked as unknown are refined during execution as the type checker observes which operations are performed on the data. As described above, unknown types are introduced for locations where values of different types stored may be stored. They are also used when the target or libraries contain incomplete type information, which may occur if they are compiled without generating debugging information, if they are linked with hand-written assembly code, and so on.

The type checker also overrides `malloc`, `free`, and other memory management routines with versions that update and test the shadow memory in the obvious ways. When the interpreted program makes a system call, the interpreter copies the arguments from the virtual registers into the processor's registers and then performs the standard kernal trap. The typechecker installs built-in instrumentation callbacks to check the validity of the argument types prior to the system call and to set the type of the return value afterwards. The most common 30 system calls have been instrumented to date.

Instruction Analysis Routines. Each instruction analysis routine type checks all occurrences of a specific instruction opcode in the target execution stream. The interpreter provides the routines with the locations of the source and destination operands. Each routine

1. checks that the source operands are allocated and initialized, and that the destination is allocated;
2. checks that load and store instructions and indirect addressing modes only dereference data of type `pointer`;
3. checks the types of the source operands and computes the result type; and

4. updates the shadow memory to reflect the destination's new type, if applicable.

We elaborate on the third and fourth steps for a representative instruction. The type checker begins step 3 by extracting the type information for the sources from the shadow memory. A table is then indexed to determine the result type. To illustrate this process, we consider the instruction `addl` SRC`,` DST. This instruction adds SRC to DST, storing the result in DST. Both operands are four byte values, as indicated by the suffix `l` in the instruction name. For example, in `addl %eax, 4(%ebp)`, the SRC operand is in register `%eax` and the DST operand is located at the address stored in register `%ebp`, plus 4.

The following table computes the result type for the operation, based on the types of SRC and DST. For simplicity, we include only a few of the possible operand types.

`addl` SRC`,` DST

SRC	DST			
	`int8`	`int32`	`pointer`	`unk32`
`int8`	`error`	`error`	`error`	`error`
`int32`	`error`	`int32`	`pointer`	`unk32`
`pointer`	`error`	`pointer`	`error`	`unk32`
`unk32`	`error`	`int32`	`pointer`	`unk32`

The type checker generates a warning whenever a table lookup returns `error`. In this example, the operands may be two integers or an integer and a pointer, but not two pointers. If DST is unknown, the result stays unknown. If SRC is unknown, the result type will be the type of DST. These heuristics for unknown types are not sound, but they reduce the false alarm rate when precise type information is not available for the operands. To aid in debugging, the type checker reports the stack trace and relevant memory and register values' types for each warning. If debugging information is available, the stack trace includes the source's file name and line number.

We show type compatibility tables for the four byte `mov` instruction and the `lea` instruction below. The `lea` sets DST to be the address of the SRC. These two instructions are insensitive to the original type of DST.

`movl` SRC`,` DST

SRC	DST
`int8`	`error`
`int32`	`int32`
`pointer`	`pointer`
`unk32`	`unk32`

`leal` SRC`,` DST

SRC	DST
`int8`	`pointer`
`int32`	`pointer`
`pointer`	`pointer`
`unk32`	`pointer`

Before returning control to the interpreter, the type checker writes the result type into the shadow memory for DST. If the new result type is different from the current type of DST and DST is invariant, the checker generates a warning. Otherwise, the initialized form of the result type is written into the shadow

memory. Since this type may have a different size than what was there previously, the type checker assigns an unknown type to any partially overwritten type encodings immediately before and after the shadow memory for DST.

The Function Call Analysis Routine. When the interpreter invokes the analysis routine for the `call` instruction, the type checker first maps the target address of the call to the corresponding function and fetches its precomputed parameter and local variable template blocks. The type checker then compares the types of the arguments on the stack against the types in the parameter template block, reporting any mismatches. It also copies the type information for the local variable template block into the shadow memory at the appropriate offset from the frame pointer for the new function's activation record. Local variables begin as uninitialized. Full function checking can not be done if the function has a variable number of arguments or uses a non-standard activation record, which may occur when a compiler employs certain optimizations, such as tail-call elimination.

Reducing False Alarms. The initial version of our type checker reported false alarms on some common compiler idioms for the x86 architecture. For example, the `gcc` compiler may emit an `xor` instruction to clear a register containing a pointer. The compiler also uses the `lea` instruction to perform addition in certain cases. To avoid generating false alarms in situations like these, we relaxed the typing restrictions in the instruction type tables. In the case of `xor`, which originally used the same table as `add` above, we permitted two pointer operands, as long as they are the same storage location. For the `lea` instruction, we deviated from the table presented above by setting the result type to `int32` if the result value is between negative one million and one million. Numbers in this range are much more likely to be integers than addresses.

Another common source of false alarms is low-level library routines in `libc`. Handwritten assembly language implementing some of the string functions is particularly problematic because it performs integer operations on sequences of four bytes. We did not wish to relax the type rules to the point where these operations are accepted because it would weaken the checking too much. Instead, we provide a way for the programmer to supply the type checker with a list of function names and specific lines of code for which no warnings should be reported. By default, warnings are turned off for the most problematic 15 functions in `libc`, including `memcpy`, `strlen`, and `tzset`. Even though warnings are not reported for these functions, they still update the shadow memory in the expected way.

5 Evaluation

Error Detection. We begin by describing our experiences applying the Hobbes type checker to the student projects from an undergraduate compilers class at Williams College. The assignments implement a compiler for a subset of the C language. Each assignment contains 3000–6000 lines of C code, plus a 3000 line parsing library, and they use the `libc` string, file, and memory routines.

Table 1. Errors found by the Hobbes type checker in a set of compiler assignments from an undergraduate class at Williams College.

Program	LOC	Unallocated	Uninitialized	Type Error	False Alarms
`p1`	5,600	1	2	2	1
`p2`	4,033	1	2	1	1
`p3`	3,571	2	3	0	1
`p4`	4,260	1	1	1	1
`p5`	4,671	2	2	1	1

Table 1 summarizes the results of running each assignment on 15 sample inputs. The table shows the number of accesses to unallocated or uninitialized memory, type errors, and false alarms reported by our tool. An error reported multiple times on different inputs is only counted once in the table. In addition, Hobbes suppresses duplicate warning messages and cascading warnings caused by an error reported earlier in a run. For example, if a program reads an uninitialized value, later warnings on that memory location or the value that was read are not reported.

The type checker reported memory errors in all five programs. The causes of these errors include calling `free` on the address of a global variable, accessing memory after it was deallocated, and incorrectly assuming that a routine in the parsing library initialized fields of a structure returned to the client.

The type checker also caught a number of type errors in the programs. In `p1`, two type errors were found. First, due to incorrect pointer arithmetic, the program overwrote an integer stored in memory with a pointer value. When that memory location was later read and multiplied by an integer, the type checker reported a type mismatch on the operands of the multiply instruction. Purify would not have caught this error since the bad pointer arithmetic would always yield a location allocated to the program. In addition, a function in `p1` passed a pointer as an argument to a function declared to take an `int` as a parameter. Inside the body of the function, the integer was cast back to a pointer and dereferenced. The type checker reported the mismatch between parameter and argument type. This code does not work properly on systems where an `int` is too small to hold a pointer, but the compiler did not warn of the problem because the programmer had not written a prototype for the function being called. A similar mistake was found in `p5`. The remaining two type errors, in `p2` and `p4`, were caused by improper uses of unions similar to Example 1 in Section 2.

The type checker erroneously reported one additional type error in each of the five programs. Each program implemented a hashtable with pointer values for keys. In each program, the function to compute hash codes generated a warning because it performed arithmetic on a pointer.

Clearly, false alarms posed no serious impediment to using the Hobbes type checker on the compiler assignments. To further explore the impact of false alarms on the utility of the Hobbes type checker, we also checked a number of larger, robust UNIX utilities. In general, the false alarm rate was acceptable. For

Table 2. Performance measurements for the SPECint 2000 benchmark. All times are in seconds and are the average of three stable runs. The Ratio columns indicate performance slowdowns relative to the Base Time.

Program	Base	Interpreter		Instrumented		MemCheck		TypeCheck	
	Time	Time	Ratio	Time	Ratio	Time	Ratio	Time	Ratio
`164.gzip`	3.0	147.9	49	219.6	73	268.0	89	470.0	157
`175.vpr`	3.1	279.6	90	334.1	108	385.1	124	532.5	172
`176.gcc`	2.3	93.5	41	136.6	59	164.5	72	286.0	124
`181.mcf`	0.4	8.6	22	12.0	30	15.0	38	21.0	53
`186.crafty`	5.5	336.8	61	473.9	86	592.8	108	1030.3	187
`197.parser`	5.4	178.2	33	257.3	48	311.4	58	486.1	90
`252.eon`	3.9	297.5	76	366.5	94	483.1	124	681.9	175
`254.gap`	1.4	52.9	38	76.5	55	97.8	70	160.6	115
`255.vortex`	9.5	472.4	50	687.9	72	889.5	94	1396.6	147
`256.bzip2`	13.9	587.6	42	791.3	57	930.4	67	1953.0	141
`300.twolf`	0.4	15.75	39	21.7	59	25.7	64	44.2	111
median			42		59		72		141

example, running `ls` with a number of different command line options netted a total of 8 spurious warnings, all of which were caused by the use of system calls not yet handled by Hobbes[1]. Several runs of `grep` generated spurious warnings, but only about uses of memory management routines in the `obstack` library, which implements a dynamic memory manager to be used in place of `malloc` and `free`. Since the `obstack` routines affect the allocation status of memory, they require special handling to be treated correctly by Hobbes and other run-time analysis tools [4]. Even in programs with much higher false alarm rates, their causes could usually be tracked to only a few problematic code sequences. Hobbes reported approximately 60 and 300 spurious warnings for runs of `vi` and `bash`, respectively. A large fraction of these false alarms are attributed to unhandled system calls, hash functions, and a small number of other code sequences.

We also verified that Hobbes could catch many classes of errors by running it on a test suite of programs with deliberate errors, all of which Hobbes found.

Hobbes catches a number of errors earlier when code is compiled without optimizations. Without optimizations, all local variables (and many intermediate values) reside on the stack, where they are marked invariant. Thus, errors can be caught as soon as an invalid value is written to a variable. In contrast, optimized code uses registers, which are not marked invariant, more heavily.

Performance. We applied the Hobbes checker to the SPECint 2000 benchmarks to evaluate the performance of our tool. Table 2 shows execution times and slowdowns for the interpreter and the interpreter instrumented with three different tools: a tool with empty analysis routines, a memory checker similar to Purify, and the type checker. All measurements are the average of three stable

[1] Note that without the suppression techniques for code in `libc` described in the previous chapter, this number would be higher.

runs on a dual-processor 1 GHz Pentium III machine with 1 GB of main memory running the Redhat Linux 2.4.9-smp kernel. We omit the 253.perlbmk benchmark because it creates new processes to do most of the computation and does not accurately reflect the impact of using the interpreter. The interpreter incurs a slowdown of 42 times over the base time. Most of this time is spent decoding the x86 instruction stream. Installing empty analysis routines for all opcodes increases the slowdown from 42 to 59. The interpreter spends the additional time storing registers and setting up activation records for the analysis routines.

MemCheck, a memory checker using a Purify-style checking algorithm, maintains *allocated* and *initialized* bits for each byte of memory. Unlike Purify's approach, MemCheck also shadows the registers with similar information to catch uses of uninitialized data. The memory checker increases the slowdown from 59 to 72. A slowdown of 13 relative to the base time is consistent with our own experience using a tool for Alpha executables based on binary modification and with reported measurements of Purify [3,10].

The Hobbes type checker runs roughly 140 times slower than normal execution, versus a slowdown of 59 for the empty analysis routines. The type checker has not been optimized for speed, and there are several significant ways to improve the performance of our prototype. Each instrumentation function typically checks memory safety first and then type safety. While this separation of tasks keeps the implementation straightforward, the two steps duplicate a nontrivial amount of work. We believe that restructuring the code to eliminate this overlap and further optimizing shadow memory operations will substantially improve performance. Additional improvements are also obtainable by switching from an interpreter to a binary translator and performing static analysis to reduce the number of instructions that must be instrumented. Finally, Hobbes is primarily a tool for testing a system, when performance is less important than correctness.

6 Related Work

Many projects have focused on identifying errors in C programs. We first describe other dynamic tools, and then a few static tools that target low-level code.

Purify [3], described in Section 1, was the first widely used memory access checker. Hobbes tracks a superset of the information tracked in Purify's shadow memory and is capable of identifying the same class of memory access errors. Memory errors that result from earlier type errors will be caught sooner in our system since Hobbes identifies them at the time of the type error. Valgrind is a more recent implementation of a Purify-like checker for Linux binaries [12].

Other memory access checkers change the representation of pointers in the target program to include capabilities [15,6,1,4]. For example, Austin et al. [1] extends the standard pointer representation to include a base and bounds for the block being referenced. Compiler-inserted code checks this extra information at each memory access. Such capability-based approaches can catch errors that Purify (and Hobbes) miss, such as when illegal pointer arithmetic yields a reference to some valid piece of memory. However, they are not compatible with standard

compiled C code. Jones and Kelly [4] store pointer base and bounds information separately, thereby achieving a higher degree of backward compatibility.

Patil and Fisher [11] demonstrated that it is sometimes possible to perform program checking in parallel with the target program execution. They present a memory access checker that incurs a slowdown as low as 10% by using a second processor to check the correctness of pointer operations.

C-Cured [10] employs a type inference scheme to statically determine which pointers in the target are used safely and which may be used improperly. Run-time checks are then inserted to check operations involving potentially unsafe pointers. C-Cured uses an extended pointer representation for these checks. This combination of static and dynamic analysis prevents memory access errors and slows down most programs by less than a factor of two, but reliance on non-standard pointer representations limits its effectiveness in some situations.

Loginov et al. [7] present a run-time type checker that uses a shadow memory similar to ours. However, they use source-to-source translation to embed the checking and maintenance code into the target. Thus, they cannot effectively check or track types through functions in compiled libraries, and they handle only programs written entirely in C. These problems also exist in several other run-time type checkers, such as Saber C [5]. Their tool is faster than Hobbes because it instruments only source-level expressions, and not every assembly-language instruction. On standard benchmarks, their tool caused roughly a 50-fold slowdown. We believe that switching to binary translation for Hobbes would eliminate most of this performance difference. A reasonable balance between precision and performance could also be obtained by inserting source code checks wherever possible and binary code checks when source code is not available or external libraries are used.

Several recent studies present static analysis techniques for C and assembly programs that would be very useful to incorporate into Hobbes. For example, Chandra and Reps devised physical type checking to check casts between different structures [2]. They characterize safe casts and define structural subtyping for C by considering the physical layout of structures. Their checking tool can successfully identify potentially unsafe casts in large programs [13]. Xu et al. [16] focus on the related problem of inferring a valid typing for a compiled program to ensure type safety before executing it. They employ abstract interpretation to construct a static approximation of the types of registers and memory at each program point. In addition, Mycroft presents a way to reconstruct C structure declarations from their use in assembly code using type inference [9]. These last two techniques would be particularly useful for reconstructing type information in situations where it is not readily available to Hobbes. Morrisett et al. [8] present a type system for x86 assembly language, but it is very different than the one underlying the Hobbes type checker because it was designed to support compilation from a type-safe high-level language, and not from C.

7 Conclusions and Future Work

Program analysis tools to identify defects in code written in unsafe languages are necessary to improve the reliability of many software systems. The Hobbes type checker can identify a large class of type errors in such systems. While our initial experiments demonstrate the effectiveness of the Hobbes methodology, we would like to improve two key aspects of our system.

Performance. Although the Hobbes interpreter provides a reasonable first prototype, implementing the type checker with a binary translation tool would significantly improve performance. Additional performance gains can also be obtained by eliminating the need to instrument every instruction. For example, static analysis could identify code fragments that are guaranteed to be type safe or that do not modify the program type state.

Precision. We would like to incorporate type inference techniques similar to those of Mycroft [9] and Xu et al. [16] to improve precision when full debugging information is unavailable. In addition, we believe that distinguishing different pointer types and identifying boundaries between structure fields and array elements would allow the Hobbes type checker to find some classes of errors sooner than it currently does. We have designed extended type encodings for this information, but we have not yet evaluated how best to use it.

References

1. T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 290–301, 1994.
2. S. Chandra and T. W. Reps. Physical type checking for C. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 66–75, 1999.
3. R. Hasting and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter Usenix Conference*, 1992.
4. R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the Third International Workshop on Automated Debugging*, pages 13–26. Linkoping University Electronic Press, 1997.
5. S. Kaufer, R. Lopez, and S. Pratap. Saber-C: an interpreterbased programming environment for the C language. In *Proceedings of the Summer Usenix Conference*, pages 161–171, 1988.
6. S. C. Kendall. Bcc: run–time checking for C programs. In *Proceedings of the Usenix Summer Conference*, 1983.
7. A. Loginov, S. H. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. In *Proceedings of the Conference on Fundamental Approaches to Software Engineering*, pages 217–232, 2001.
8. J. G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In *Types in Compilation*, pages 280–52, 1998.
9. A. Mycroft. Type-based decompilation. In *Proceedings of the European Symposium of Programming*, pages 208–223, 1999.

10. G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 128–139, 2002.
11. H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Software–Practice and Experience*, 27(1):87–110, January 1997.
12. J. Seward. Valgrind, an open-source memory debugger for x86-GNU/Linux, August 2002. Available from `http://developer.kde.org/~sewardj/`.
13. M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. W. Reps. Coping with type casts in C. In *Proceedings of ESEC/FSE '99*, pages 180–198, 1999.
14. A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205, 1994.
15. J. L. Steffen. Adding run-time checking to the portable C compiler. *Software – Practice and Experience*, 22(4):305–316, 1992.
16. Z. Xu, T. Reps, and B. P. Miller. Typestate checking of machine code. In *Proceedings of the European Symposium of Programming*, pages 335–351, 2001.