Low-Level Library Analysis and Summarization[⋆]

Denis Gopan[1] and Thomas Reps[1,2]

[1] University of Wisconsin
[2] GrammaTech, Inc.
{gopan,reps}@cs.wisc.edu

Abstract. Programs typically make extensive use of libraries, including dynamically linked libraries, which are often not available in source-code form, and hence not analyzable by tools that work at source level (i.e., that analyze intermediate representations created from source code). A common approach is to write *library models* by hand. A library model is a collection of function stubs and variable declarations that capture some aspect of the library code's behavior. Because these are hand-crafted, they are likely to contain errors, which may cause an analysis to return incorrect results.

This paper presents a method to construct summary information for a library function automatically by analyzing its low-level implementation (i.e., the library's binary).

1 Introduction

Static program analysis works best when it operates on an entire program. In practice, however, this is rarely possible. For the sake of maintainability and quicker development times, software is kept modular with large parts of the program hidden in libraries. Often, commercial off-the-shelf (COTS) modules are used. The source code for COTS components and libraries (such as the Windows dynamically linked libraries) is not usually available.

In practice, the following techniques are used to deal with library functions:

- **stop at library calls:** this approach reduces analysis coverage and leads to incomplete error detection;
- **treat library calls as identity transformers:** this approach is generally unsound; furthermore, this approach is imprecise because the analysis models a semantics that is different from that of the program;
- **define transformers for selected library calls:** this approach is not extensible: new transformers must be hardcoded into the analyzer to handle additional calls;
- **use hand-written source-code stubs that emulate some aspects of library code:** while this approach is both sound and extensible, the process of crafting stubs is usually time-consuming and error-prone.

In this paper, we describe a static-analysis tool that automatically constructs summaries for library functions by analyzing their low-level implementation (i.e., binary code). A library function's summary consists of a set of *error triggers* and a set of *summary transformers*. Error triggers are assertions over the program state that, if satisfied at the call site of the function, indicate a possibility of program failure during the function's invocation. Summary transformers specify how the program state is affected by the function call: they are expressed as *transfer relations*, i.e., the relations that hold among the values of global variables and function parameters *before* and *after* the call.

[⋆] Supported by ONR under grant N00014-01-1-0796 and by NSF under grants CCF-0540955 and CCF-0524051.

To use the function summaries, a client analysis must approximate the set of program states that reach the call site of a library function. The analysis should report an error if the approximation contains states that satisfy an assertion that corresponds to some error trigger. Summary transformers are applied as follows: the “before” values of global variables and function parameters are restricted to those that reach the call site; the restricted transfer relation is projected onto the “after” values to yield an approximation for the set of program states at the function’s return point.

Our work makes the following contributions:

- It provides a way to create library summaries automatically, which frees tool developers from having to create models for standard libraries by hand.
- The library summaries obtained have applications in both verification tools and bug-finding/security-vulnerability-detection tools, and thus help in both kinds of code-quality endeavors.
- The library summaries obtained by our tool could be used in client analysis tools that work on either source code or low-level code (i.e., assembly code, object code, or binary executable code). In particular, they satisfy the needs of many source-code bug-finding analyses, which propagate symbolic information through the program, including the amount of memory allocated for a buffer, the offset of a pointer into a corresponding buffer, the length of a string, etc. [25, 10].
- In some cases, the tool might allow static analysis to be carried out more efficiently. That is, the application/library division provides a natural modularity border that could be exploited for program-analysis purposes: typically, many applications link against the same library; summarizing the functions in that library obviates the need to reanalyze the library code for each application, which could improve analysis efficiency. (See §5 for a discussion of other work that has had the same motivation.)

During development, application code is changed more frequently than library code. Because an application can be analyzed repeatedly against the same set of library summaries, it is possible to recoup the cost of applying more sophisticated analyses, such as polyhedral analysis [8], for library summarization.

Some may argue against our choice of analyzing the low-level implementation of library functions: programs link against any possible implementation of the library, possibly across different platforms. Thus, it would be desirable to verify programs against more abstract function summaries derived, for instance, from library specifications. Below, we list some of the reasons why we believe that constructing library functions from low-level implementations deserves attention.

- Formal library specifications are hard to get hold of, while a low-level implementation for each supported platform is readily available.
- Even if formal specification is available, there is no easy way to verify that a particular library implementation conforms to the specification.
- The analysis of an actual library implementation may uncover bugs and undesirable features in the library itself. For instance, while summarizing memory-management functions, we discovered that the *libc* implementation that came with Microsoft Developer Studio 6 assumes that the direction flag, the x86 flag that specifies the direction for string operations, is set to *false* on entry to the library. This can be a security vulnerability if an adversary could control the value of the direction flag prior to a subsequent call to `memcpy`.

The remainder of the paper is organized as follows: §2 provides an overview of our goals, methods, and results obtained. §3 discusses the individual steps used to generate summary information for a library function. §4 summarizes two case studies. §5 discusses related work.

2 Overview

We use the function `memset` as the running example for this paper. The function is declared as follows:

```
void * memset ( void * ptr, int value, size_t num );
```

Its invocation sets the first *num* bytes of the block of memory pointed to by *ptr* to the specified value (interpreted as an unsigned char). The value of *ptr* is returned.

In this paper, we address two types of memory-safety errors: buffer overruns and buffer underruns. Typically, analyses that target these types of errors propagate allocation bounds for each pointer. There are many ways in which this can be done. We use the following model. Two auxiliary integer variables are associated with each pointer variable: $alloc_f$ is the number of bytes that can be safely accessed "ahead" of the address stored in the pointer variable, $alloc_b$ is the number of bytes that can be safely accessed "behind" the address stored in the pointer variable. We believe that this scheme can be easily interfaced with other choices for representing allocation bounds. We use dot notation to refer to allocation bounds of a pointer variable, e.g., $ptr.alloc_f$.

Analysis goals. The function summary specifies how to transform the program state at the call site of the function to the program state at its return site. Also, it specifies conditions that, if satisfied at the call site, indicate that a run-time error is possible during the function call. Intuitively, we expect the summary transformer for the `memset` function to look like this (for the moment, we defer dealing with memory locations overwritten by *memset* to §3.2):

$$ret = ptr \wedge ret.alloc_f = ptr.alloc_f \wedge ret.alloc_b = ptr.alloc_b, \qquad (1)$$

where ret denotes the value that is returned by the function. We expect the sufficient condition for the buffer overflow to look like this:

$$num \geq 1 \wedge ptr.alloc_f \leq num - 1. \qquad (2)$$

The goal of our analysis is to construct such summaries automatically.

Analysis overview. Fig. 1 shows the disassembly of `memset` from the C library that is bundled with Visual C++.[3] Observe that there are no explicit variables in the code; instead, offsets from the stack register (**`esp`**) are used to access parameter values. Also, there is no type information, and thus it is not obvious which registers hold memory addresses and which do not. Logical instructions and shifts, which are hard to model numerically, are used extensively. Rather than addressing all these challenges at once, the analysis constructs the summary of a function in several phases.

[3] We used Microsoft Visual Studio 6.0, Professional Edition, *Release* build.

```
00401070   mov  edx, dword ptr [esp + 12]  edx ← count
00401074   mov  ecx, dword ptr [esp + 4]   ecx ← ptr
00401078   test edx, edx
0040107A   jz   004010C3                   if(edx = 0) goto 004010C3
0040107C   xor  eax, eax                   eax ← 0
0040107E   mov  al, byte ptr [esp + 8]     al ← (char)value
00401082   push edi
00401083   mov  edi, ecx                   edi ← ecx
00401085   cmp  edx, 4
00401088   jb   004010B7                   if(edx < 4) goto 004010B7
0040108A   neg  ecx                        ecx ← −ecx
0040108C   and  ecx, 3                     ecx ← ecx & 3
0040108F   jz   00401099                   if(ecx = 0) goto 00401099
00401091   sub  edx, ecx                   edx ← edx − ecx
00401093   mov  byte ptr [edi], al         *edi ← al
00401095   inc  edi                        edi ← edi + 1
00401096   dec  ecx                        ecx ← ecx − 1
00401097   jnz  00401093                   if(ecx ≠ 0) goto 00401093
00401099   mov  ecx, eax                   ecx ← eax
0040109B   shl  eax, 8                     eax ← eax << 8
0040109E   add  eax, ecx                   eax ← eax + ecx
004010A0   mov  ecx, eax                   ecx ← eax
004010A2   shl  eax, 10h                   eax ← eax << 16
004010A5   add  eax, ecx                   eax ← eax + ecx
004010A7   mov  ecx, edx                   ecx ← edx
004010A9   and  edx, 3                     edx ← edx & 3
004010AC   shr  ecx, 2                     ecx ← ecx >> 2
004010AF   jz   004010B7                   if(ecx = 0) goto 004010B7
004010B1   rep stosd                 while(ecx ≠ 0) { *edi ← eax; edi++; ecx--; }
004010B3   test edx, edx
004010B5   jz   004010BD                   if(edx = 0) goto 004010BD
004010B7   mov  byte ptr [edi], al         *edi ← al
004010B9   inc  edi                        edi ← edi + 1
004010BA   dec  edx                        edx ← edx − 1
004010BB   jnz  004010B7                   if(edx ≠ 0) goto 004010B7
004010BD   mov  eax, dword ptr [esp + 8]   eax ← ptr
004010C1   pop  edi
004010C2   retn                            return
004010C3   mov  eax, dword ptr [esp + 4]   eax ← ptr
004010C7   retn                            return
```

Fig. 1. The disassembly of `memset`. The rightmost column shows the semantics of each instruction using a C-like notation.

Intermediate Representation (IR) recovery. First, *value-set analysis (VSA)* [1, 2] is performed on the disassembled code to discover low-level information: variables that are accessed by each instruction, parameter-passing details, and, for each program point, an overapproximation of the values held in the registers, flags, and memory locations at that point. Also, VSA resolves the targets of indirect control transfers.

In x86 executables, parameters are typically passed via the stack. The register **`esp`** points to the top of the stack and is implicitly updated by the **`push`** and the **`pop`** instructions. VSA identifies numeric properties of the values stored in **`esp`** and maps offsets from **`esp`** to the corresponding parameters. To see that this process is not trivial, observe that different offsets map to the same parameter at addresses *0x4010BD* and *0x4010C3*: at *0x4010BD* an extra 4 bytes are used to account for the push of **`edi`** at *0x401082*.

Numeric Program Generation. VSA results are used to generate a numeric program that captures the behavior of the library function. The primary challenge that is addressed in this phase is to translate non-numeric instructions, such as bitwise operations and shifts, into a program that numeric analysis is able to analyze. Bitwise operations are used extensively in practice to perform certain computations because they are typically more efficient in terms of CPU cycles than corresponding numeric instructions. The ubiquitous example is the use of **`xor`** instruction to initialize a register to zero. In Fig. 1, the **`xor`** at *0x40107C* is used in this way.

The generation phase also introduces the auxiliary variables that store allocation bounds for pointer variables. A simple type analysis is performed to identify variables and registers that may hold addresses. For each instruction that performs address arithmetic, additional statements that update corresponding allocation bounds are generated. Also, for each instruction that dereferences an address, a set of numeric assertions are generated to ensure that memory safety is not violated by the operation. The assertions divert program control to a set of *error program points*.

Numeric Analysis and Summary Construction. The generated numeric program is fed into an off-the-shelf numeric analyzer. We use a numeric analysis that, instead of approximating sets of reachable program states, approximates program transfer functions. That is, for each program point, the analysis computes a function that maps an approximation for the set of initial states at the entry of the program to an approximation for the set of states that arise at that program point. The numeric-analysis results are used to generate a set of error triggers and a set of summary transformers for the library function. The transfer functions computed for program points corresponding to the return instructions form a set of summary transformers for the function. Error triggers are constructed by projecting transfer functions computed for the set of error program points onto their domains.

The summary obtained for `memset`. Memset uses two loops and a "**`rep stosd`**" instruction, which invokes a hardware-supported loop. The "**`rep stosd`**" instruction at *0x4010B1* is the workhorse; it performs the bulk of the work by copying the value in `eax` (which is initialized in lines *0x40107C–0x40107E* and *0x401099–0x4010A5* to contain four copies of the low byte of `memset`'s $value$ parameter) into successive 4-byte-aligned memory locations. The loops at *0x401093–0x401097* and *0x4010B7–0x4010BB* handle any non-4-byte-aligned prefix and suffix. If the total number of bytes to be initialized is less than 4, control is transfered directly to the loop at *0x4010B7*.

The application of our technique to the code in Fig. 1 yields exactly the summary transformer we conjectured in Eqn. (1). The situation with error triggers is slightly more complicated. First, observe that there are three places in the code where the buffer is accessed: at addresses *0x401093*, *0x4010B1*, and *0x4010B7*. Each access produces a separate error trigger:

$$\begin{array}{ll} \mathit{0x401093}: & num \geq 4 \wedge ptr.alloc_f \leq 2 \\ \mathit{0x4010B1}: & num \geq 4 \wedge ptr.alloc_f \leq num - 1 \\ \mathit{0x4010B7}: & num \geq 1 \wedge ptr.alloc_f \leq num - 1 \end{array}$$

Note that the first trigger is stronger than the one conjectured in Eqn. (2): it gives a constant bound on $alloc_f$; furthermore, the bound is less than 3, which is the smallest bound implied by the conjectured trigger. The issue is that the instruction at *0x401093* accesses at most three bytes. In case $ptr.alloc_f$ is equal to 3, `memset` will generate a

```
                  memset($(ptr, value, num)$)
00401070              $edx \leftarrow count;$
00401074              $ecx \leftarrow ptr; ecx.alloc_f \leftarrow ptr.alloc_f; ecx.alloc_b \leftarrow ptr.alloc_b;$
00401078-7A           if$(edx = 0)$ goto $L5;$
0040107C-82           ...
00401083              $edi \leftarrow ecx; edi.alloc_f \leftarrow ecx.alloc_f; edi.alloc_b \leftarrow ecx.alloc_b;$
00401088              if$(edx < 4)$ goto $L3;$
0040108A              $ecx \leftarrow -ecx;$
0040108C              $ecx \leftarrow ?;$ assume$(0 \leq ecx \leq 3);$
0040108F              if$(ecx = 0)$ goto $L2;$
00401091              $edx \leftarrow edx - ecx;$
00401093          L1: assert$(edi.alloc_f >= 1);$ assert$(edi.alloc_b >= 0);$
00401095              $edi \leftarrow edi + 1;\ edi.alloc_f \leftarrow edi.alloc_f - 1;\ edi.alloc_b \leftarrow edi.alloc_b + 1;$
00401096              $ecx \leftarrow ecx - 1;$
00401097              if$(ecx \neq 0)$ goto $L1;$
00401099-A5       L2: ...
004010A7              $edx.rem_4 = ?;\ edx.quot_4 = ?;$
                      assume$(0 \leq edx.rem_4 \leq 3);$ assume$(edx = 4 \times edx.quot_4 + edx.rem_4);$
                      $ecx \leftarrow edx;\ ecx.quot_4 \leftarrow edx.quot_4;\ ecx.rem_4 = edx.rem_4;$
004010A9              $edx \leftarrow edx.rem_4;$
004010AC              $ecx \leftarrow ecx.quot_4;$
004010AF              if$(ecx = 0)$ goto $L3;$
004010B1              assert$(edi.alloc_f >= 4 \times ecx);$ assert$(edi.alloc_b >= 0);$
                      $edi \leftarrow edi + 4 \times ecx;$
                      $edi.alloc_f \leftarrow edi.alloc_f - 4 \times ecx;\ edi.alloc_b \leftarrow edi.alloc_b + 4 \times ecx;$
004010B3-B5           if$(edx = 0)$ goto $L4;$
004010B7          L3: assert$(edi.alloc_f >= 1);$ assert$(edi.alloc_b >= 0);$
004010B9              $edi \leftarrow edi + 1;\ edi.alloc_f \leftarrow edi.alloc_f - 1;\ edi.alloc_b \leftarrow edi.alloc_b + 1;$
004010BA              $edx \leftarrow edx - 1$
004010BB              if$(edx \neq 0)$ goto $L3;$
004010BD          L4: $eax \leftarrow ptr;\ eax.alloc_f = ptr.alloc_f;\ eax.alloc_b \leftarrow ptr.alloc_b;$
004010C2              return $eax, eax.alloc_f, eax.alloc_b;$
004010C3          L5: $eax \leftarrow ptr;\ eax.alloc_f = ptr.alloc_f;\ eax.alloc_b \leftarrow ptr.alloc_b;$
004010C7              return $eax, eax.alloc_f, eax.alloc_b;$
```

Fig. 2. The numeric program generated for the code in Fig. 1; parts of the program that are not relevant for the summary construction are omitted.

buffer overrun at one of the other memory accesses. The other two triggers are similar to the trigger conjectured in Eqn. (2) and differ only in the value of num.

3 Library Code Analysis

3.1 Intermediate Representation Recovery

The IR-recovery phase recovers intermediate representations from the library's binary that are similar to those that would be available had one started from source code. For this phase, we use the CodeSurfer/x86 analyzer that was developed jointly by Wisconsin and GrammaTech, Inc. This tool recovers IRs that represent: control-flow graphs (CFGs), with indirect jumps resolved; a call graph, with indirect calls resolved; information about the program's variables; possible values of pointer variables; sets of used, killed, and possibly-killed variables for each CFG node; and data dependences. The techniques employed by CodeSurfer/x86 do not rely on debugging information being present, but can use available debugging information (e.g., Windows .pdb files) if directed to do so.

The analyses used in CodeSurfer/x86 (see [1, 2]) are a great deal more ambitious than even relatively sophisticated disassemblers, such as IDAPro [15]. At the technical level, they address the following problem: *Given a (possibly stripped) executable E (i.e., with all debugging information removed), identify the procedures, data objects, types, and libraries that it uses, and, for each instruction I in E and its libraries, for each interprocedural calling context of I, and for each machine register and variable V, statically compute an accurate over-approximation to the set of values that V may contain when I executes.*

Variable and Type Discovery. One of the major stumbling blocks in analyzing executables is the difficulty of recovering information about variables and types, especially for aggregates (i.e., structures and arrays).

When debugging information is absent, an executable's data objects are not easily identifiable. Consider, for instance, a data dependence from statement `a` to statement `b` that is transmitted by write/read accesses on some variable `x`. When performing source-code analysis, the programmer-defined variables provide us with convenient compartments for tracking such data manipulations. A dependence analyzer must show that `a` defines `x`, `b` uses `x`, and there is an `x`-def-free path from `a` to `b`. However, in executables, memory is accessed either directly—by specifying an absolute address—or indirectly—through an address expression of the form "[*base* + *index* $\times$ *scale* + *offset*]", where *base* and *index* are registers and *scale* and *offset* are integer constants. It is not clear from such expressions what the natural compartments are that should be used for analysis. Because, executables do not have *intrinsic* entities that can be used for analysis (analogous to source-level variables), a crucial step in IR recovery is to identify variable-like entities.

The variable and type-discovery phase of CodeSurfer/x86 [2], recovers information about variables that are allocated globally, locally (i.e., on the run-time stack), and dynamically (i.e., from the heap). An iterative strategy is used; with each round of the analysis—consisting of aggregate structure identification (ASI) [19, 2] and value-set analysis (VSA) [1, 2]—the notion of the program's variables and types is refined. The net result is that CodeSurfer/x86 recovers a set of proxies for variables, called *a-locs* (for "abstract locations"). The a-locs are the basic variables used in the method described below.

3.2 Key Concepts of Numeric Program Generation

The generation of a numeric program is the central piece of our work. We strive as much as possible to generate a sound representation of the binary code.[4] The target language is very simple: it supports assignments, assumes, asserts, if-statements, and gotos. The expression "?" selects a value non-deterministically. The condition "*" transfers control non-deterministically.

Translating x86 instructions. Due to space constraints, we only describe several translation issues that are particularly challenging. The numeric instructions, such as **`mov`**, **`add`**, **`sub`**, **`lea`**, etc., are directly translated into the corresponding numeric statements:

[4] Currently, we assume that numeric values are not bounded. In the future, we hope to add support for bounded arithmetic.

e.g., the instruction "`sub edx,ecx`" at *0x401091* in Fig. 1 is translated into numeric statement $edx \leftarrow edx - ecx$.

The bitwise operations and shifts typically cannot be precisely converted into a single numeric statement, and thus pose a greater challenge. Several numeric statements, including ifs and assumes, may be required to translate each of these instructions. At first we were tempted to design universal translations that would work equally well for all possible contexts in which the instruction occurs. In the end, however, we noticed that these instructions, when used for numeric computations, are only used in a few very specific ways. For instance, bitwise-and is often used to compute the remainder from dividing a variable by a power of two. The instruction "`and ecx,3`" at *0x40108C* in Fig. 1 is used to compute ecx **mod** 4. The translation treats these special cases with precision; other cases are treated imprecisely, but soundly. The instruction "**`and`** $op_1,\ op_2$" is translated into "$op_1 \leftarrow ?$; **assume**$(0 \leq op_1 \leq op_2)$;" if op_2 is an immediate operand that has a positive value; otherwise, it is translated into "$op_1 \leftarrow ?$;".

Recovering conditions from the branch instructions. An important part of numeric program generation is the recovery of conditional expressions. In the x86 architecture, several instructions must be executed in sequence to perform a conditional control transfer. The execution of most x86 instructions affects the set of flags maintained by the processor. The flags include the *zero flag*, which is set if the result of the currently executing instruction is zero, the *sign flag*, which is set if the result is negative, and many others. Also, the x86 architecture provides a number of control-transfer instructions each of which performs a jump if the flags are set in a specific way. Technically, the flag-setting instruction and the corresponding jump instructions do not have to be adjacent and can, in fact, be separated by a set of instructions that do not affect the flags (such as **`mov`** instruction.

We symbolically propagate the expressions that affect flags to the jump instructions that use them. Consider the following sequences of instructions and their translation:

cmp eax,ebx	
mov ecx,edx	$ecx \leftarrow edx;$
jz label	**if**$(eax - ebx = 0)$ **goto** $label;$

We derive a flag-setting expression $eax - ebx$ from the **`cmp`** instruction; the **`mov`** instruction does not affect any flags; the **`jz`** instruction transfers control to `label` if the zero flag is set, which can only happen if the expression $eax - ebx$ is equal to zero. Note, however, that if the intervening move affects one of the operands in the flag-setting expression, that expression is no longer available at the jump instruction. This can be circumvented with the use of a temporary variable:

cmp eax,ebx	
mov eax,edx	$temp \leftarrow eax - ebx;\ eax \leftarrow edx;$
jz label	**if**$(temp = 0)$ **goto** $label;$

Allocation bounds. As we mentioned above, each variable var that may contain a memory address is associated with two auxiliary variables that specify allocation bounds for that address. The auxiliary variable $var.alloc_f$ specifies the number of bytes following the address that can be safely accessed; the auxiliary variable $var.alloc_b$ specifies the number of bytes preceding the address that can be safely accessed. These auxiliary variables are central to our technique: the purpose of numeric analysis is to find constraints on the auxiliary variables that are associated with the function's parameters and return value. These constraints form the bulk of the function summaries.

The updates for the auxiliary variables are generated in a straightforward way. That is, the translation of the **`mov`** instruction contains assignments for the corresponding allocation bounds. The instructions **`add`**, **`sub`**, **`inc`**, **`dec`**, and **`lea`**, as well as the x86 string-manipulation instructions, are translated into affine transformations on variables and their associated allocation bounds.

The auxiliary variables are used to generate memory-safety checks: checks for buffer overflows and checks for buffer underflows. We generate these checks for each indirect memory access that does not access the current stack frame. As mentioned in §3.1, general indirect memory accesses in x86 instructions have the form "[*base* + *index* $\times$ *scale* + *offset*]", whose base and index are registers and scale and offsets are constants. Let *size* denote the number of bytes to be read or written. The following checks are generated:

– Buffer-overflow check: **assert**$(base.alloc_f \geq index * scale + \textit{offset} + \textit{size})$
– Buffer-underflow check: **assert**$(base.alloc_b + index * scale + \textit{offset} \geq 0)$

The checks generated for the x86 string-manipulation instructions, such as **`stos`** and **`movs`** are only slightly more complicated and are omitted for brevity.

Type Analysis. Maintaining allocation bounds for all variables is unnecessarily expensive. For this reason, we only associate allocation bounds with variables that can hold memory addresses. To identify this set of variables, we construct an *affine-dependence graph (ADG)*: a graph in which the nodes correspond to program variables and the edges indicate that the value of the destination variable is computed as an affine transformation of the value of the source variable. The construction of the ADG is straight-forward: e.g., instruction "**`mov`** `foo, bar`" generates an edge from the node that corresponds to variable `bar` to the node that corresponds to `foo`, etc. To determine the set of pointer variables, we start with nodes that correspond to variables that are used as *base pointers* in memory-safety checks and mark as pointers all the variables whose corresponding nodes are reached by a backward traversal through the graph.

Note that the precision of the ADG does not affect the soundness of the overall analysis: if some dependences are not present in the graph, then some allocation bounds will not be tracked and the overall analysis will be less precise. If some non-existing dependences are present in the graph, then some useless allocation bounds will be tracked and the analysis will be slowed down.

In contrast to variables, which keep the same type throughout their lifetime, registers are reused in different contexts, and can have a different type in each context. Limiting each register to a single node in the ADG generates many "spurious" dependences because all of the contexts in which the register is used are collapsed together. Thus, when constructing the ADG, we create a separate node for each register's live-range.

Handling integer division and remainders. Memory functions, such as `memset`, rely heavily on integer division and remainder computations to improve the efficiency of memory operations. In low-level code, the quotient and remainder from dividing by a power of two are typically computed with a shift-right (**`shr`**) instruction and a bitwise-and (**`and`**) instruction, respectively. In Fig. 1, the two consecutive instructions at *0x4010A9* establish the property: $edx_0 = 4 \times ecx + edx$, where edx_0 denotes the value contained in `edx` before the instructions are executed. This property is essential for inferring precise error triggers for memory accesses at *0x4010B1* and *0x4010B7*. However, polyhedral analysis is not able to handle integer division with sufficient precision.

We overcome this problem by introducing additional auxiliary variables: each variable var that may hold a value for which both a quotient and remainder from division by k are computed is associated with two auxiliary variables $var.quot_k$ and $var.rem_k$, which denote the quotient and the remainder, respectively. To identify such variables, we use the ADG: we look for the nodes that are reachable by backward traversals from both the quotient and remainder computations. The auxiliary variables are associated with all of the nodes that are visited by the traversals up to the first shared node. For the above example, the starting point for the "quotient" traversal is the use of **`ecx`** at *0x4010AC*, and the staring point for the "remainder" traversal is the use of **`edx`** at *0x4010A9*: at these points, we generate assignments that directly use the corresponding auxiliary variables. The first shared node is the use of **`edx`** at *0x4010A7*: at that point, we generate numeric instructions that impose semantic constraints on the values of auxiliary variables (see Fig. 2). The intermediate updates for the auxiliary variables are generated in a straightforward way. Polyhedral analysis of the resulting program yields precise error triggers for both memory accesses.

Modeling the environment. The goal of our technique is to synthesize the summary of a library function by looking at its code in isolation. However, library functions operate in a larger context: they may access memory of the client program that was specified via their parameters, or they may access global structures that are internal to the library. The IR-recovery phase has no knowledge of either the contents or the structure of that memory: they are specific to the client application. As an example, from the IR-recovery perspective, `memset` parameter ptr may contain any memory address. Thus, from the point of view of numeric-program generation, a write into $*ptr$ may potentially overwrite any memory location: local and global variables, a return address on the stack, or even the code of the function. As the result, the generated numeric program, as well as the function summary derived from it, will be overly conservative (causing the client analysis to lose precision).

We attempt to generate more meaningful function summaries by using *symbolic constants* to model memory that cannot be confined to a specific a-loc by the IR-recovery phase. A unique symbolic constant is created for each unresolved memory access. From numeric-analysis perspective, a symbolic constant is simply a global variable that has a special auxiliary variable $addr$ associated with it. This auxiliary variable represents the address of a memory location that the symbolic constant models. If the memory location may hold an address, the corresponding symbolic constant has allocation bounds associated with it. We illustrate this technique in §4.

3.3 Numeric Analysis and Summary Generation

Our numeric analyzer is based on the Parma Polyhedral Library (PPL) and the WPDS++ library for weighted pushdown systems (WPDSs) and supports programs with multiple procedures, recursion, global and local variables, and parameter passing. The analysis of a WPDS yields, for each program point, a *weight*, or abstract state transformer, that describes how the program state is transformed on all the paths from the entry of the program to that program point. Linear-relation analysis [8] is encoded using weights that maintain two sets of variables: the *domain* describes the program state at the entry point; the *range* describes the program state at the destination point. The relationships between the variables are captured with linear inequalities. Given a weight computed for some program point, its projection onto the range variables approximates the set of states that are reachable at that program point. Similarly, its projection onto the set of domain variables approximates the precondition for reaching that program state.

Function summaries are generated from the numeric-analysis results. Summary transformers are constructed from the weights computed for the program points corresponding to procedure returns. Error triggers are constructed by back-projecting weights computed for the set of error program points.

4 Case Studies

We used our technique to generate summaries for library functions `memset` and `_lseek`. The IR-recovery and numeric-program generation was done on 1.83GHz Intel Core Duo T2400 with 1.5Gb of memory. The numeric analysis was done on 2.4GHz Intel Pentium 4 with 4Gb of memory.

The summary obtained for `memset`. The detailed description of `memset`, as well as the analysis results, were given in §2 and §3. It took 70 seconds to both execute the IR-recovery phase and generate a numeric program for `memset`. The resulting numeric program has one procedure with 8 global variables and 11 local variables. The numeric analysis took 1 second.

The summary obtained for `_lseek`. the function `_lseek` moves a file pointer to a specified position within the file. It is declared as follows:

```
off_t _lseek(int fd, off_t offset, int origin);
```

fd is a file descriptor; *offset* specifies the new position of the pointer relative to either its current position, the beginning of the file, or the end of the file, based on *origin*.

A recurring memory-access pattern in `_lseek` is to read a pointer from a global table and then dereference it. Fig. 3 shows a portion of `_lseek` that contains a pair of such memory accesses: the first **`mov`** instruction reads the table entry, the second dereferences it. The registers **`ecx`** and **`edx`** hold the values *fd*$/32$ and *fd* **mod** 32, respectively. The global variable *uNumber* gives the upper bound for the possible values of *fd*. Symbolic constants mc_1 and mc_2 model the memory locations accessed by the first and second **`mov`** instructions, respectively. Our technique synthesizes the following buffer-overrun trigger for the second **`mov`** instruction:

$$\text{0x424DE0} \leq mc_1.addr \leq \text{0x424DE0} + (\textit{uNumber} - 1)/8 \;\wedge\; mc_1.alloc_f <= 251$$

The above trigger can be interpreted as follows: *if any of the addresses stored in the table at 0x424DE0 point to a buffer of length that is less than 252 bytes, there is a possibility of a buffer-overrun error*. The error trigger is sufficient for a client analysis to implement sound error reporting: if the client analysis does not know the allocation bounds for pointers in the table at *0x424DE0*, it should emit an error report for this trigger at the call site to `_lseek`. However, we hope that the summary generated by our technique for the library-initialization code will capture the proper allocation bounds for the pointers in the table at *0x424DE0*. Thus, the analysis will not emit spurious error reports. The error triggers for other memory accesses look similar to this one.

The analysis took about 70 seconds to recover intermediate representation and generate a numeric program. The generated program has 41 global variables (22 of which

```
mov   eax, dword ptr [4×ecx + 0424DE0h]
```
$\textbf{assume}(mc_1.addr = \text{0x424DE0} + 4 * ecx);$

$eax \leftarrow mc_1;\ eax.alloc_f = mc_1.alloc_f;\ eax.alloc_b = mc_1.alloc_b;$

```
movsx ecx, byte ptr [eax + 8×edx + 4]
```
$\textbf{assert}(eax.alloc_f \leq 8 * edx + 5);\ \textbf{assert}(eax.alloc_b + 8 * edx + 4 \geq 0);$

$\textbf{assume}(mc_2.addr = eax.alloc_b + 8 * edx + 4 \geq 0);\ ecx \leftarrow\ mc_2;$

Fig. 3. Symbolic memory modeling: the symbolic constants mc_1 and mc_2 model the memory location accessed by **mov** and **movsx** instructions, repsectively.

are used for symbolic memory modeling) and contains three procedures with 21, 8, and 2 local variables, respectively. The numeric analysis of the program took 117 seconds.

5 Related Work

Summary functions have a long history, which goes back to the seminal work by Cousot and Halbwachs on linear-relation analysis [8] and the papers on interprocedural analysis of Cousot and Cousot [7] and Sharir and Pnueli [24]. Other work on analyses based on summary functions includes [16, 20, 3], as well as methods for pushdown systems [11, 4, 5, 21], where summary functions arise as one by-product of an analysis.

A substantial amount of work has been done to create summary functions for alias analysis or points-to analysis [18, 26, 14, 6, 22], or for other simple analyses, such as lock state [27]. Those algorithms are specialized for particular problems; more comprehensive approaches include the work on analysis of program fragments [23], componential set-based analysis [12], and use of SAT procedures [27].

Some of the work cited above explicitly mentions separately compiled libraries as one of the motivations for the work. Although the techniques described in the aforementioned papers are language-independent, all of the implementations described are for source-code analysis.

Guo et al. [13] developed a system for performing pointer analysis on a low-level intermediate representation. The algorithm is only partially flow-sensitive: it tracks registers in a flow-sensitive manner, but treats memory locations in a flow-insensitive manner. The algorithm uses partial transfer functions [26] to achieve context-sensitivity, where the transfer functions are parameterized by “unknown initial values”.

Kruegel et al. [17] developed a system for automating mimicry attacks. Their tool uses symbolic-execution techniques on x86 binaries to discover attacks that can give up and regain execution control by modifying the contents of the data, heap, or stack so that the application is forced to return control to injected attack code at some point after a system call has been performed. Cova et al. [9] used this platform to apply static analysis to the problem of detecting security vulnerabilities in x86 executables. In both of these systems, alias information is not available.

In our work, we make use of a-locs (variable proxies), alias information, and other IRs that have been recovered by the algorithms used in CodeSurfer/x86 [1, 2]. The recovered IRs are used as a platform on which we implemented a relational analysis that synthesizes summary functions for procedures.

Acknowledgements. We thank D. Vitek for sharing with us his insights on the problem of creating function summaries.

References

1. G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *CC*, 2004.
2. G. Balakrishnan and T. Reps. DIVINE: DIscovering Variables IN Executables. In *VMCAI*, 2007.
3. T. Ball and S.K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *PASTE*, 2001.
4. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model checking. In *CONCUR*, 1997.
5. A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *POPL*, 2003.
6. R. Chatterjee, B.G. Ryder, and W. Landi. Relevant context inference. In *POPL*, 1999.
7. P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *Formal Descriptions of Programming Concepts*. North-Holland, 1978.
8. P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *POPL*, 1978.
9. M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In *ACSAC*, 2006.
10. N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI*, 2003.
11. A. Finkel, B.Willems, and P. Wolper. A direct symbolic approach to model checking pushdown systems. *Elec. Notes in Theor. Comp. Sci.*, 9, 1997.
12. C. Flanagan and M. Felleisen. Componential set-based analysis. In *PLDI*, 1997.
13. B. Guo, M.J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D.I. August. Practical and accurate low-level pointer analysis. In *3nd Int. Symp. on Code Gen. and Opt.*, 2005.
14. M.J. Harrold and G. Rothermel. Separate computation of alias information for reuse. *TSE*, 22(7), 1996.
15. IDAPro disassembler, http://www.datarescue.com/idabase/.
16. J. Knoop and B. Steffen. The interprocedural coincidence theorem. In *CC*, 1992.
17. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *USENIX Sec. Symp.*, 2005.
18. W. Landi and B.G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *PLDI*, 1992.
19. G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *POPL*, 1999.
20. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.
21. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. of Comp. Prog.*, 58(1–2), 2005.
22. A. Rountev and B.G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *CC*, 2001.
23. A. Rountev, B.G. Ryder, and W. Landi. Data-flow analysis of program fragments. In *FSE*, 1999.
24. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
25. D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, 2000.
26. R.P. Wilson and M.S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*, 1995.
27. Y. Xie and A. Aiken. Scalable error detection using Boolean satisfiability. In *POPL*, 2005.