SMART C: A Semantic Macro Replacement Translator for C

Matthew Jacobs E Christopher Lewis
Department of Computer and Information Science
University of Pennsylvania
{*mrjacobs,lewis*}*@cis.upenn.edu*

Abstract

*Programmers often want to transform the source or binary representations of their programs (*e.g.*, to optimize, add dynamic safety checks, or add profile gathering code). Unfortunately, existing approaches to program transformation are either disruptive to the source (hand transformation), difficult to implement (*ad hoc *analysis tools), or functionally limited (macros). We propose an extension to the C programming language called the Semantic Macro Replacement Translator (SMART C). SMART C allows for the specification of very general type-aware transformations of all operations, statements, and declarations of the C programming language without exposing the programmer to the complexities of the system's internal representations. We have implemented a prototype SMART C source-to-source translator and show its use in transforming programs for buffer overflow detection, format string vulnerability detection, and weighted call graph profiling. We show that SMART C achieves a pragmatic balance between generality and ease of use.*

1. Introduction

Programmers and users often want to transform the source or binary representations of their C language programs in particular ways. For example, the performance conscious would like to perform domain or application-specific optimization without hard coding these optimizations into the source program, thus preserving the natural (unoptimized) program logic. Programmers sometimes want to encode dynamic checks in programs to ensure certain dynamic properties (*e.g.*, that an array is not accessed beyond its bounds), and users often want to use dynamic checks to prevent potentially buggy applications from compromising user or system integrity (*e.g.*, by restricting system calls). Transformation is also used to inject instrumentation code in order to gather profile data about a program's dynamic behavior which is useful in guiding offline optimization.

A variety of techniques exists to effect such transformations on C programs, but they are each limited in that they are disruptive to the program source, beyond the reach of typical programmers, or restricted in the transformations they may describe. Hand transformation clearly suffers from the first limitation. The most powerful approach to program transformation is to augment an existing compiler, such as GCC, to build an *ad hoc* transformation tool. Unfortunately, this requires considerable effort and expertise; most programmers lack one or both of these. Binary and dynamic rewriting tools (*e.g.*, ATOM [23] and Pin [16]) are powerful, but they cannot reliably transform source-level constructs because some constructs (*e.g.*, structure field access) are not necessarily apparent at the instruction level. Aspect-oriented programming (AOP) systems [13] are easy to use, but existing AOP designs (even those applied to C) are limited in the language-level constructs that may be transformed. Finally, macro systems such as `cpp` and `m4` are simple and easy to use, but they are very limited in the transformations they may specify.

In this paper, we propose a modest extension to the C programming language called the Semantic Macro Replacement Translator for C (SMART C). Unlike token-based macros (*e.g.*, those of `cpp`), semantic macros operate on the abstract syntax of a program and are type aware. SMART C allows for the transformation of any declarative or computational element of the C language without exposing the internal representation of the compiler. As a result, programmers can freely transform variable and function declarations, statements, and even primitive operations such as arithmetic or logical operations. SMART C transformations can be predicated on both syntactic (*e.g.*, variable names) and semantic (*e.g.*, variable types) properties of the code. In addition, SMART C includes a limited form of transformation-time evaluation that balances generality and ease of use. Finally, the SMART C design preserves the spirit of the C language, introducing little new syntax and leveraging existing programmer intuition. In summary, SMART C is powerful, general, compact, and easy to use.

In order to use SMART C, a programmer defines a set of semantic macros (s-macros). S-macro expansion is guided by patterns that determine what source-level con-

```
Source code:                Transformed code:
float a, b, c;   ==>        float a, b, c;
a = b/c;                    a = b * (1.0/c);

Transformation specification:
around(FP % / FP %) {
   return tc_operand * (1.0/tc_operand2);
}
```

(a) Floating point division transformation.

```
Source code:                Transformed code:
int a[10];       ==>        struct {int value[10];
                                    bool isValid;} a;

Transformation specification:
around(decl(Integer[] %)) {
   struct{tc_type value; bool isValid;} tc_name;
   tc_body;
}
```

(b) Array declaration transformation.

Figure 1. Example SMART C transformation specification and their effect on C source.

structs (*e.g.*, floating point division and integer array declaration) are to be transformed. When a pattern specified in an s-macro matches a source-level construct, the macro is expanded. For example, suppose we wish to transform floating point division into multiplication of the numerator and the reciprocal of the denominator. The s-macro to achieve this appears in Figure 1(a). The pattern "`FP % / FP %`" indicates that the s-macro should match all division operations that have floating point operands (with any names). The `around` keyword indicates that the macro body should replace the division expression (versus being inserted before or after it). The macro body computes the product as a function of the values of the numerator expression (`tc_operand`) and denominator expression (`tc_operand2`). Figure 1(b) illustrates the transformation of all integer array declarations to structure declarations containing the original integer array and a boolean flag.

This work makes the following contributions. We present the design of a semantic macro system for C that is simple (leveraging programmer intuition, requiring little new syntax, avoiding exposing intermediate representations), powerful (useful and interesting transformations may be specified), and concise (requiring very little SMART C code to achieve useful transformations). We describe our SMART C implementation and show its utility in three different application contexts.

2. SMART C Design

SMART C macro expansion is a source-to-source transformation guided by a set of user-specified transformation specifications. Each transformation specification consists of both semantic macros (s-macros) and (transformation-local) auxiliary code and data declarations required by the s-macros. An s-macro consists of (i) a pattern describing the expressions, statements, or declarations to be transformed, (ii) a body containing code, and (iii) a modifier describing how the matched entity is to be transformed. We call the untransformed and transformed programs the *base code* and *target code*, respectively. A matching entity in the base code is called a *match site*. Below we introduce the components of s-macros, but space constraints preclude complete, manual-style presentation.

2.1. Patterns

An s-macro *pattern* is an abstract description of C source-level primitives and can describe any expression, statement, or declaration (variable or function). The matching process matches on both the primitive (*e.g.*, addition or function call) and the types/names of the operands. For example, the s-macro in Figure 1(a) only matches division of floating point operands. Patterns have three components, each of which plays a role in pattern matching: (i) a type specifying the type of an operand (*e.g.*, a floating point number), (ii) a specification of the name of an operand (*e.g.*, a function called `malloc`), and (iii) a C language primitive (*e.g.*, division or variable declaration). Below, we describe each pattern component.

Pattern types. A *pattern type* specifies the data type of an operand in a pattern. In addition to all the C primitive data types, pattern types may include any user-defined data types or any of the additional (shaded) types appearing in the type hierarchy in Figure 2. These additional types are only available in SMART C patterns. (*i.e.*, they cannot be used in any C code). Each pattern type in Figure 2 matches all C types in the nodes beneath it, thus allowing for the concise specification of a set of related types (*e.g.*, all signed integers of any precision). Derived types such as structures, pointers or arrays may be created from these primitive types. In addition, the `Any` type can be used to specify any possible type, including derived types. The `Any` type may also be refined, as in the case of the pattern type `Any *`, which matches pointers to any type. The syntax of pattern types is borrowed directly from the C language.

Pattern names. A *pattern name* specifies the name of an operand in a pattern. This includes variable and function names as well as literals. Names may include the wildcard character `%`, which matches any number of characters. For example, the pattern name `%alloc` would match both `malloc` and `calloc`. Pattern names can also represent literals by using quotes. The literals matched will depend

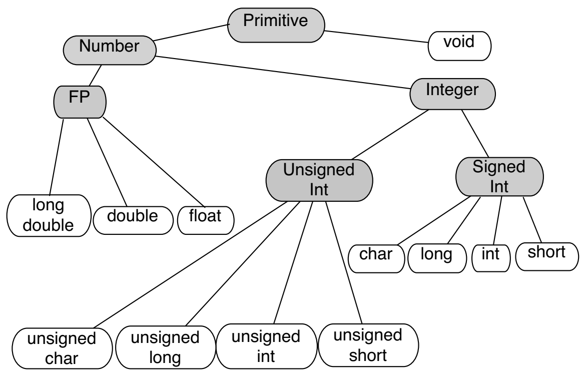

Figure 2. Primitive type hierarchy used for matching in SMART C.

upon the pattern type (above). For example, if the pattern type is Integer, then `"1"` refers to the integer literal 1, while `"%"` refers to any integer literal. Although general regular expressions would provide additional flexibility beyond our wildcard character, we have not found a need for this generality.

Pattern primitives. A *pattern primitive* specifies the C language primitive that the pattern should match. Pattern primitives are built from names and types (specifying the name and types of the primitive operands, where applicable), and can represent C expressions, statements, or declarations. The syntax of pattern primitives, again, is borrowed from the C language itself. For example, the pattern in the s-macro of Figure 1(a) matches division primitives; the two type/name operands specify that we should only match if the division operands are floating point values of any precision with any name.

SMART C provides expression patterns for all arithmetic, comparison, and logical operators, in addition to other primitive operations of the C language (dereference, address-of, array access, structure access, function call). SMART C provides pattern primitives representing sets of related C primitives, thus allowing for the specification of more generic patterns. For example, the `Binop` primitive matches all binary operations and `Unop` matches all unary operations. These abstract primitives are further refined based on their potential return types (*e.g.*, arithmetic versus logical binary operations).

Statement patterns (`if`, `ifelse`, `switch`, `while`, `dowhile`, `for`) do not require any pattern types or names (*i.e.*, an `if` pattern will match all `if` statements independent of the type of its predicate expression or structure of its body). We have not found a need for more selective matching.

Declaration patterns match either variable or function declaration. There our four kinds of variable declaration patterns: `decl`, `globaldecl`, `localdecl`, and `formaldecl`. The `decl` pattern matches all variable declarations, while the remaining three match global, local, and formal parameter declarations, respectively.

SMART C also contains the boolean patterns `not`, `and`, and `or` that combine multiple patterns, which have the natural interpretation. This admits, for example, patterns that match additions in which *either* operand is a pointer type.

2.2. Modifiers

An s-macro *modifier* describes how the s-macro body should be inserted at a match site. The modifier may take on the values of `before`, `after`, or `around` (terms borrowed from aspect-oriented programming [13]), indicating whether the body of the s-macro should be inserted before, after, or instead of, respectively, the matching expression, statement, or declaration. The method by which bodies are inserted is described in the next section.

2.3. Bodies

An s-macro *body* defines the code that is to be inserted (according to the modifier) into the program (at the match site). The body simply consists of C code, augmented with SMART C-specific variables and syntax that allow the body to be parameterized based on the context in which it is inserted. *Context variables* describe properties of the matched expression, statement, or declaration; and *transformation-time control statements* allow for the body to be customized based on these properties.

Context variables. Context variables are place holders for values, names, declarations, code, or operations that are part of the matching expression, statement, or declaration. Context variables allow the body code to be parameterized based on properties of the match site. Each context variable is recognized by the "`tc_`" prefix ("this context"). Since context variables appear only in SMART C code, the usage of variables beginning with "`tc_`" in C remains unrestricted.

Table 1 shows the context variables available for each type of pattern (although we will see that the context variables in parentheses are not defined for all bodies). The context variables that are available in a particular body are determined by the properties of the match site, which are apparent from the pattern associated with the body. The `tc_func` context variable always denotes the function in which the match site is located.

Expression patterns are created out of operands and an operation. The value of a matching expression can be accessed via `tc_expr`. Each operand can, in general, be an arbitrary expression of any type. The type and value of the first matched operands are accessed through the `tc_type` and `tc_operand` context variables. When matching bi-

Pattern	Available Context Variables
Expression	tc_expr, tc_operand, tc_type, (tc_operand2), (tc_type2), tc_operation, tc_rettype, (tc_args), (tc_numargs), tc_func
Statement	tc_stmt, tc_expr, (tc_expr2), (tc_expr3), tc_block, (tc_block2), tc_func
Declaration	tc_declscope, tc_name, tc_type, tc_decl, tc_body, (tc_args), tc_func

Table 1. Context Variables.

nary expressions, the tc_type2 and tc_operand2 context variables are also available. The operation is accessed via the tc_operation() function. The result type of the operation is denoted by tc_rettype. If the expression is a function call, then the argument list is via tc_args. This is an array of arguments, each possessing type and value information. For example, the context variables tc_args[1].expr and tc_args[1].type are place holders for the value of the value and type of the second argument to the function. The number of arguments to the function is denoted by the context variable tc_numargs.

Statement patterns match control-flow statements. The statement matched by the pattern may be accessed by tc_stmt. In each control-flow statement, there is an expression or series of expressions to be evaluated and control goes to a block of statements based upon the expression value. These expressions are denoted by the tc_expr context variables, and the code blocks are denoted by the tc_block context variables. For example, an if-else statement possesses one expression to evaluate, and a choice of two code blocks to be conditionally executed. The context expressions tc_expr, tc_block, and tc_block2 will be available to use in the transformation body.

Declaration patterns match variable or function declarations. Each variable declaration is associated with a statement block, logically giving each declaration its own scope. The context variable tc_declscope is used to denote the combination of the declaration and its associated code block. The declaration itself is denoted by tc_decl and the associated code block by tc_body. The type and name of the declared variable are denoted by tc_type and tc_name. For function declarations, the context variables tc_type and tc_name denote the return type and function name respectively. In this case, the function body is represented by tc_body, and the function prototype by tc_decl. As in the function call case, the arguments can be accessed though the tc_args context variable.

It is often useful to have the textual representation of that which a context variable represents. For example, suppose tc_operand represents some match site variable counter. It is useful to make the string "counter" available to the transformation body. This is achieved by preceding any context variable with a dollar sign. In this example, we could include the following code in our body: fprintf(log, $tc_operand).

Transformation-time control statements. Like context variables, transformation-time control constructs are a SMART C-specific construct that can appear within s-macro bodies. The three possible transformation-time control constructs are IF-ELSE, FOR, and SWITCH. These statements are formed in the same way as their C counterparts. However, they are restricted in what they may contain so that they may be evaluated at transformation-time.

The IF and IF-ELSE constructs selectively include code in the target program based on the value of a predicate (*i.e.*, the predicate is evaluated at transformation time and either the then or the else statement—if one exists—is included in the target code). The predicate must be a transformation-time constant expression, consisting of C literals, context variables, any C operation, and the functional subset of the string library. A simple example adapted from Engler [10] appears in Figure 3. If a non-reentrant function is called from within a signal handler (by convention named with a "sig_" prefix), the transformation results in an error and program termination. This s-macro matches all calls to non-reentrant functions (just nonreentrant() in this example). SMART C also provides transformation-time display (PRINTF()) and termination (EXIT()) operations that could be used in this example to report the same error at transformation time. PRINTF() and EXIT() cannot be nested within C control flow, but appearing within transformation-time control flow is allowed.

Unlike the C for loop, the SMART C FOR loop must iterate a transformation-time constant number of iterations, requiring that (i) the initializer must be a simple assignment of an integer transformation-time constant, (ii) the comparator must compare the induction variable to an integer transformation-time constant, and (iii) the incrementor may increment or decrement the induction variable by an integer transformation-time constant. During transformation, the number of iterations is computed and the loop is fully unrolled.

Similarly, the SWITCH statement must be governed by a transformation-time constant expression. Although this expression may be any transformation-time constant expression, expressions representing C types will be particularly useful, allowing for the insertion of type-specific code during transformation. An example (also adapted from Engler [10]) exploiting both the FOR and SWITCH statements appears in Figure 4. In this example, the programmer calls an output(...) function in the base code that is type-unaware. This function call is transformed to an appropriate type-dependent printf() call.

```
before(Any nonreentrant()) {
  IF(strncmp($tc_func,"sig_",4) == 0) {
   printf("Sig handling error");
   exit(1);
  }
 }
```

Figure 3. S-macro using `IF`.

```
around(void output(...)) {
 char typeString[] = {0};
 FOR(int i=0; i<tc_numargs; i++) {
  SWITCH(tc_args[i].type) {
   CASE FP:
     strcat(typeString, "%f"); break;
   CASE Char *:
     strcat(typeString, "%s"); break;
   CASE Any *:
     strcat(typeString, "%p"); break;
   ...
   }
  }
  printf(typeString, tc_args);
 }
```

Figure 4. S-macro using `FOR` and `SWITCH`.

The above transformation-time control allows s-macro bodies to be parameterized based on the context of the match site, yet they remain easy to reason about for both the compiler (enabling static type checking) and programmer. We have considered more general computation (*e.g.*, including transformation-time variables), but we have not yet found a pressing need for it.

2.4. Discussion

SMART C presents a simple model of transformation to programmers. In order to leverage the user's intuition from the C language and obviate the need to learn internal representations, programmers can only define patterns that match single primitives in the language (*e.g.*, binary operations, function calls, declarations, etc.). SMART C users can build patterns that match the operations or operands in complex expressions (*e.g.*, `a + b + c`), but they cannot match the whole expression. SMART C sacrifices this generality for three reasons: (i) the resulting language is much simpler from a user's perspective, (ii) the practical limitation is minimal because multi-primitive patterns are fragile in that they are tied to particular programming idioms, and (iii) we find that the resulting language is still quite useful and powerful.

S-macro bodies are type checked to ensure that if the pattern they contain match any base code construct, the resulting target code with be type correct (as far as the C language is concerned). S-macros may be type checked independent of the code to which they are applied because types (or classes of types) are statically apparent from the s-macro itself. The general procedure for type checking is identical to C type checking except that context variables can take on multiple types. This set of types is determined from the pattern associated with the s-macro. The type checker conservatively assumes the context variables may have any of these types and rejects programs that may violate C's typing rules.

3. Implementation

This section describes how pattern matching and s-macro expansion are realized. We also summarize the current implementation status.

3.1. Pattern Matching

Pattern matching determines which match sites will be transformed by SMART C. First, the SMART C pattern matcher walks though the list of declarations in the program, attempting to match each against any of declaration patterns. If a match is found, both the declaration and associated code block are transformed accordingly. An example of this can be found in Figure 7(a). The SMART C pattern matcher then walks through the AST and examines each statement. If this statement matches a statement pattern, the transformation is applied at this time. Finally, the expression matching is performed by doing a bottom-up walk over the syntax tree of this statement. Each operation is checked for a match against all expression patterns, and a transformation is applied if a match is found. In each case, the pattern is considered a match only when the pattern primitive matches as well as any expression, name or type information embedded within the pattern matches as well.

Each transformation specification (consisting of sets of s-macros) operates independently of all other transformation specifications. They are composed by applying one to the result of another. The order of application is specified by the user. This simple strategy allows transformation specifications to be modular. For example, a programmer could write transformation specifications that add null pointer checks and log all function calls. These could be written separately and independently, and applied in either order.

For each transformation specification, any source code primitive may match and be transformed by at most one s-macro. To resolve ambiguities (*i.e.*, multiple patterns matching at a single match site), the user-specified order of s-macros in a transformation specification is used. The first matching s-macro "wins." To achieve this, SMART C processes the s-macros in the order they appear in the source file. This gives the programmer flexibility to resolve ambiguities.

3.2. Code Transformation

Once SMART C has found a pattern match, it must determine how to transformation the match site. SMART C must

Source code:
```
int *ptr, result;
ptr = (int *)
  malloc(sizeof(int));
result = *ptr;
```
⟹

Transformed code:
```
int *ptr, *temp, result;
ptr = (int *)
 malloc(sizeof(int));
if (ptr == 0)
 error("NULL DEREF");
temp = ptr;
result = *temp;
```

S-macro:
```
before(*Any* % ) {
   if (tc_operand == 0) error("NULL DEREF")
}
```

(a) Before transformation on dereference expression.

Source code:
```
int a, b;
a = b * 2;
```
⟹

Transformed code:
```
int a, b;
a = b << 1;
```

S-macro:
```
around(Integer % * "2"){
   return tc_operand << 1;
}
```

(b) Around transformation on multiply expression.

Figure 5. Example SMART C expression transformations.

Source code:
```
if (x > 0)
  foo(x);
```
⟹

Transformed code:
```
printf("Test %s",
 "x > 0");
if (x > 0)
  foo(x);
```

S-macro:
```
before(if) {
   printf("Test %s", $tc_expr);
}
```

(a) Before transformation on if statement.

Source code:
```
for(i=0; i < 10; i++)
  {printf("i=%d", i);
}
```
⟹

Transformed code:
```
i = 0;
while (i < 10) {
  printf("i=%d", i);
  i++; }
```

S-macro:
```
around(for){
 tc_expr;
 while(tc_expr2) {
   tc_block;
   tc_expr3; }
}
```

(b) Around transformation on for statement.

Figure 6. Example SMART C statement transformations.

resolve both the transformation-time control constructs and context variables that appear in the transformation body. All of the context variables may be statically determined by examining the match site. Each context variable is replaced by the appropriate variable, type, operation, expression, declaration or code block, as described above.

Expression transformation. Expression transformation is governed by the `before`, `after`, or `around` s-macro modifiers. Since the expression is part of a statement, it is desirable to isolate this expression from the rest of the statement for transformation purposes. To do this, SMART C creates a temporary variable of the expression's type and replaces the appearance of the expression at the match site with this temporary variable.

If the transformation modifier is `before` or `after`, a new statement is created that assigns the matched expression into the temporary variable from above. At this point, the macro body (with context variables replaced, as above) is inserted before or after the newly created statement. An example is shown in Figure 5(a).

If the modifier is `around`, the expression will be replaced by the result of the macro body. In this case, the macro body must end with a return statement which specifies a value of the same type as the matched expression. This expression is assigned into the temporary variable described above, and the macro body is inserted before this statement. An example is shown in Figure 5(b).

Statement transformation. A statement may be transformed by a s-macro using the `before`, `after`, or `around` modifiers. If the modifier is `before` or `after`, the macro body is inserted directly before or after the match site. If the modifier is `around`, then the statement is replaced with the macro body. Examples of each are shown in Figure 6.

Declaration transformation. A declaration may be transformed by a s-macro using the `before`, `around`, or `after` modifiers. This pattern matches on the declaration, and may replace or add to the declaration and the code in the scope associated with matched declaration. The modifiers specify where code is added so that a `before` s-macro will add declarations before the match site and code before the associated scope. The `after` and `around` cases are

Source code:
```
int a;
foo();
```
⟹ Transformed code:
```
int a;
add_object(a);
foo();
del_object(a);
```
S-macro:
```
around(decl(Integer %)) {
   tc_decl;
   add_object(tc_name);
   tc_body;
   del_object(tc_name);
}
```
(a) Around transformation on variable declaration.

Source code:
```
foo(int a, float b)
 { do_stuff(); }
```
⟹ Transformed code:
```
foo(int a, float b) {
 init_table();
 do_stuff();
 delete_table(); }
```
S-macro:
```
around(decl(Any %(...))){
   tc_decl {
    init_table();
    tc_body;
    delete_table(); }
}
```
(b) Around transformation on function declaration.

Figure 7. Example SMART C declaration transformations.

similar. An example of changing a variable declaration has already been presented in Figure 1(b). Examples demonstrating another transformation of a variable declaration and the body of a function declaration are shown in Figure 7.

3.3. Implementation Status

SMART C is implemented as a source-to-source translator. SMART C takes as input one file containing an arbitrary number of transformation specifications, and any files containing source code that the user wishes to transform. SMART C outputs the transformed versions of each of these source files.

Our SMART C implementation has been built using the C-Breeze compiler infrastructure [15]. C-Breeze provides a number of predefined phases and allows for custom phases to be built. Our implementation uses four phases. First, the source code and the transformation specifications are parsed into an AST. Next they are dismantled using C-Breeze's built-in dismantler, resulting in a three-address-code-like intermediate representation. The next phase walks through the dismantled AST to search for C constructs which match a s-macro pattern. Upon a pattern match, the transformation body is expanded and applied to the match site as discussed in the previous section. The final phase "undismantles" the code, converting it to a higher-level, more readable version, which serves as the output.

4. Applications

This section demonstrates the use of SMART C across a wide variety of application domains. Space constraints preclude the presentation of complete transformation specifications, so we instead describe only the most important s-macros.

4.1. Buffer Overflow Detection

Recently, there has been a great amount of work done to make C programs safe with respect to buffer overflows. Buffer overflows are possible in C because no explicit bounds-checking occurs. Hackers have found numerous ways to construct malicious input which subverts control of the system by overwriting critical control areas through the use of carefully constructed strings which overwrite unchecked buffers in C code. There are a variety of solutions to this problem; we will examine two here. Both of the solutions have been proposed and implemented previously by modifying a compiler. We will show that SMART C provides a way to offer this functionality without having to deal with the complexity of compiler internals.

```
typedef struct {
  void * value;
  void * base;
  unsigned size;
  enum {Heap, Local, Global} storageClass;
  int capability;
} SafePtr;

around(decl(Any * %)) {
    SafePtr tc_name;  //make all ptrs into SafePtrs
    tc_body;          //keep body the same
}
```

Figure 8. SafeC s-macro that transforms pointer declaration to fat pointer declaration translation.

The first buffer overflow detection mechanism we consider is SafeC [2]. SafeC is a program transformation that changes the representation of pointers to "fat pointers," which are C structures that contain spatial and temporal attributes. This transformation requires that every use of a pointer must be transformed to update or check this pointer metadata appropriately.

In SMART C, these transformations are easy to express. The SafeC system has been implemented via SMART C using fifteen transformations, requiring 150 lines of code (excluding the C runtime library routines). The most important SMART C-based SafeC transformation is the conversion

```
around(Void* %alloc(...)) {
    tc_rettype ptr;
    ptr = tc_expr; //ptr stores result of malloc
    add_object(ptr, tc_args[0].expr); // add to table
    return ptr; //return address
}
```

Figure 9. CRED s-macro that transforms `malloc` call to allocation plus object insertion.

of pointer declarations to fat pointers. The transformation specification to achieve this appears in Figure 8. Each operation that performs pointer arithmetic is modified to update the correct `SafePtr` components. Assignments to pointers must also reflect the new `SafePtr` representation. Pointer creation through a `malloc` call or the '&' operation must generate pointer attributes to place in the `SafePtr` representation. The transformation of pointer dereferencing does not change the semantics of dereference, but inserts checks to ensure the spatial and temporal attributes of the pointer result in a valid dereference. Finally, each function body must be transformed to include prologue and epilogue code to generate and discard scoping information, which is used by the `SafePtr` representation to update and verify its temporal attributes.

Next, we consider the buffer overflow detection technique proposed by Ruwase and Lam [19] called CRED (C Range Error Detector). CRED does not change the pointer representation; rather, it keeps object metadata in an auxiliary runtime table, and checks each pointers' value against this table to verify its validity. This involves adding bounds information about each object in the program to the table, and updating it appropriately when objects are deallocated.

These transformations are easy to describe in SMART C. The object table, out-of-bounds (OOB) table, and helper functions which provide an interface to modify the tables are provided at the top level of the transformation specification. Objects must be inserted into the object table for each object (non-pointer) declaration and each call to `malloc`. The code for the `malloc` case is shown in Figure 9. These objects are deleted upon the termination of a scope or a call to `free`, respectively. S-macros are also necessary to update the tables appropriately and to perform checks upon pointer dereference. The transformation for binary operations where the first operand is a pointer is shown in Figure 10.

4.2. Format String Vulnerability Detection

Programs written in C are also subject to format string attacks. These attacks are achieved by giving the program a string which will be passed to `printf` as a format string. This string can be formed in a particular way to use it's % directives to write an arbitrary value to memory. Format

```
around(and(BinOp(Any * %, Any %),
  not(BinOp(Any * %, Any * %))) {
  //matches all expr of form : ptr (op) non-ptr
    tc_type base;
    tc_rettype result, retval;
    result = tc_operation(
      get_oob(tc_operand), tc_operand2);
      //look up ptr in OOB, do original operation
    if (check_ptr(result)) retval=result;
      //nothing to do if object is inbounds
    else
      retval = add_oob(result, base_obj(tc_operand));
      //if OOB, add (address,base) to OOB
    return retval; //return result of computation }
```

Figure 10. CRED s-macro that transforms a binary operation to use object and OOB table information.

```
before(Int printf(...) {
  char * formatStr = tc_args[0].expr;
  int numPercent = parse(formatStr);
  if (numPercent > tc_numargs-1) {
    printf("Attack detected!");
    exit(1);
  }
}
```

Figure 11. FormatGuard s-macro that transforms calls to `printf` to check the number of arguments.

string vulnerabilities are possible in C because no checking is performed to ensure that the format string contains % directives that match the number and types of further arguments passed to `printf`.

FormatGuard [7] detects many forms of this attack by dynamically ensuring that the number of arguments to `printf()` are the same as the number of `%` directives in the format string. While FormatGuard uses `cpp` to effect this transformation, SMART C admits a much simpler specification (Figure 11).

Alternatively, format string vulnerabilities can be statically detected. Shankar et al. use type qualifiers (specifically `tainted`) to discover when format strings are derived from user-supplied input [20]. SMART C can generate code to dynamically compute the same thing. To achieve this, taintedness information is maintained and propagated for each string, and all strings used as format strings are checked to ensure that they are not tainted. This transformation is similar to Safe C, in that SMART C changes the representation of strings to include a taintedness bit, and each access to a string must be transformed in the obvious manner.

4.3. Weighted Call Graph Construction

The call graph is a useful tool in application profiling. Call graphs encode the control flow between functions by counting how many times and from where each procedure gets dynamically called. Call graphs are typically constructed as the program runs and analyzed offline.

Weighted call graph construction is easily realized with SMART C. The `main` function is augmented with code to initialize a global data structure as its prologue, and code to calculate the call graph from the global data structure as its epilogue. Internally, pairs of (*caller*, *callee*) functions are stored in a hash table and associated with a count. For each occurrence of a (*caller*, *callee*) pair, the count is incremented. Each function body is transformed to begin with an update to the hash table that stores the *callee* name. Each function call is transformed to also include an update to the hash table that stores the call site. Upon termination of the program, there will be a sequence of pairs of function names (*caller*, *callee*) with an associated count. From this information, it is simple to construct the weighted call graph.

5. Related Work

Code transformation approaches may be distinguished by the representations on which they operate. Below we summarize systems that operate on the token stream, the abstract syntax tree, or the machine code of a program.

Syntax-based patterns. Transformation systems which operate on the syntactic structure of C code have more expressiveness than token-based transformations, and allow transformations to be architecture-independent, unlike binary translators. SMART C provides the ability to match and transform primitives of the C language. *MAGIK* [10] provides a library to access AST primitives and transform them. Unlike SMART C, traversals over the AST must be explicitly performed by the programmer.

Systems exist that are able to match and transform arbitrary ASTs. These transformations are sensitive to programmer idioms, and may be more brittle than transformations on primitives. Moreover, writing transformational code is harder for these systems since there are more pieces of the AST to reason about. The Code Transformation Tool (*ctt*) [4] is an example of such a system. The *Stratego* [25] system uses term rewriting to express transformations. *ASTLOG* [8] uses a Prolog variant as a transformation language. ASF+SDF is an environment for automatically constructing languages and provides facilities for their transformation [24].

Transformation systems may also operate on keywords introduced by the programmer that will expand into C code. *ASTEC* [17] is a system which operates on code that has not yet been pre-processed, and is designed to be a replacement for `cpp`. As `cpp` does, *ASTEC* matches on keywords introduced by the programmer, and is therefore inappropriate for applying transformations to preexisting base code. Likewise, the MS^2 (Meta Syntactic Macro System) [26] uses a language that may access pieces of the AST directly and use them in code expansion.

Aspect-oriented programming (AOP) [13] is a general framework for expressing crosscutting concerns in a modular fashion. The most well-known versions are *AspectJ* [12] and *AspectC++* [22]. These systems allow programmers to match and transform method calls and variable access, and refine the match sites by further matching upon dynamic control flow information. Many systems have brought AOP concepts to C, including *AspectC* [5], *c4* [27], *Aspicere* [1], *Arachne* [9] and *TinyC* [28]. These systems all provide the ability to match on function calls, and in some cases variable access and dynamic control flow information. We argue that SMART C patterns made up of all primitive operations of C allow a more expressive set of transformations than solely function calls. In fact, the dynamic control flow matching provided in some of these languages can be expressed as a transformation specification in SMART C, obviating the need for a special language construct.

Token-based patterns. Transformation tools that reason about a token representation of base code include the *cpp* and *m4* macro systems. These tools suffer from the disadvantage that no contextual information is available about the match site. Furthermore, subtle errors may be introduced due to precedence and side-effects that are not obvious from the macro code.

Binary-based patterns. Systems that operate on the binary representation of the program are designed with a different set of goals than systems which operate on ASTs. They seek to provide a machine-specific transformation capability, at the loss of semantic information (such as types), potential optimization opportunities, and portability across architectures. *ATOM* [23] and *EEL* [14] are examples of compile-time transformation systems which operate on the binary representation of the program. There are also a variety of run-time transformation systems which allow programmers hooks into the binary representation of a program, some examples include *Dynamo* [3], *Pin* [16], and *DISE* [6].

Metaprogramming. The concept of transformation-time control in SMART C (`IF-ELSE, FOR, SWITCH`) is an instance of metaprogramming. Other metaprogramming systems include Template Haskell [21] and the Scheme language [11]. Metaprogramming allows programmers to write code which produces other code. Many metaprogramming systems, such as *tcc* [18], allow this metacode to be arbitrary. In general, this makes code written in a metaprogramming language to be difficult to reason about. SMART C provides only a limited set of language constructs to produce code at transformation-time. Again, this is an example where SMART C attempts to limit the complexity of transformational code, while still maintaining enough power to express meaningful transformations.

6. Conclusion

We have introduced semantic macros to the C programming language via SMART C (Semantic Macros Replacement Transformer for C). Our SMART C extensions allow for far more transformation power than traditional C macro systems because (i) type information is used the pattern matching/replacement process, (ii) any C language primitive may be transformed, and (iii) our macro bodies are highly parameterizable. We show the use of SMART C in several practical contexts (buffer overflow detection, format string vulnerability detection, and call graph profiling), and we find that powerful transformations can very simply and succinctly be represented with SMART C.

References

[1] B. Adams and T. Tourwé. Aspect Orientation for C: Express yourself. In *Proceedings of Software-Engineering Properties of Languages and Aspect Technologies*, 2005.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of Conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[4] M. Boekhold, I. Karkowski, and H. Corporaal. Transforming and Parallelizing ANSI C Programs Using Pattern Recognition. In *Proceedings of International Conference on High-Performance Computing and Networking*, pages 673–682, 1999.

[5] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to Improve the Modularity of Path-Specific customization in operating system code. In *Foundations of Software Engineering*, 2001.

[6] M. L. Corliss, E. C. Lewis, and A. Roth. DISE: A Programmable Macro Engine for Customizing Applications. In *Proceedings of International Symposium on Computer Architecture*, pages 362–373, 2003.

[7] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic Protection From `printf` Format String Vulnerabilities. In *Proceedings of USENIX Security Symposium*, 2001.

[8] R. F. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *Proceedings of Conference on Domain-Specific Languages*, 1997.

[9] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Segura-Devillechaise, and M. Sudholt. An expressive aspect language for system applications with Arachne. In *Proceedings of International Conference on Aspect-oriented software development*, 2005.

[10] D. R. Engler. Incorporating application semantics and control into compilation. In *Proceedings of Conference on Domain-Specific Languages*, 1997.

[11] R. Kelsey, W. Clinger, and J. R. (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proc. of European Conf. on Object-Oriented Programming*, 2001.

[13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[14] J. R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of Conference on Programming Language Design and Implementation*, pages 291–300, 1995.

[15] C. Lin, S. Z. Guyer, and D. Jimenez. The C-Breeze Compiler Infrastructure. Technical Report TR-01-43, The University of Texas at Austin, November 2001.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[17] B. McCloskey and E. Brewer. ASTEC: A New Approach to Refactoring C. In *Proceedings of Foundations of Software Engineering*, 2005.

[18] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation. In *Proceedings of Conference on Programming Language Design and Implementation*, pages 109–121, 1997.

[19] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of Network and Distributed System Security Symposium*, pages 159–169, 2004.

[20] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of USENIX Security Symposium*, 2001.

[21] T. Sheard and S. P. Jones. Template Meta-programming for Haskell. In *Proceedings of Workshop on Haskell*, pages 1–16, 2002.

[22] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of International Conference on Technology of Object-Oriented Langiages and Systems*, 2002.

[23] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[24] M. van den Brand, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of Compiler Construction 2001 (CC 2001)*, LNCS. Springer, 2001.

[25] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, 2004.

[26] D. Weise and R. Crew. Programmable Syntax Macros. In *Proceedings of Conference on Programming Language Design and Implementation*, pages 156–165, 1993.

[27] M. Yuen, M. Fiuczysnki, R. Grimm, Y. Coady, and D. Walker. Making extensibility of system software practical with the C4 toolkit. In *AOSD Workshop on Software Engineering Properties of Languages and Aspect Technologies*, 2006.

[28] C. Zhang and H.-A. Jacobssen. TinyC^2: Towards Building a Dynamic Weaving Aspect Language for C. In *Foundations of Aspect-Oriented Languages*, 2003.