

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 transforma-

tions 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 time and expertise; most programmers lack one or both of these. Binary and dynamic rewriting tools (e.g., ATOM [24] and Pin [17]) 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 [14] 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. The `around` keyword indicates that the macro body should replace the division operation (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 via three case studies.

This paper is organized as follows. Section 2 defines SMART C and describes its use. Section 3 presents the implementation. Section 4 presents the application of SMART C to three problems requiring source-level transformation. The final two sections summarize related work, offer conclusions, and suggest future work.

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. S-macros consist of (i) a pattern describing the expressions, statements, or declarations to be transformed,

(ii) a body containing code, and (iii) a modifier describing how a matched expression, statement, or declaration is to be transformed via the body. Each s-macro operates by finding a pattern match with a *match site* in the *base code*; the code at the match site is transformed by the s-macro into the *target code*. 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 on 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) an expression specifying 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 of the additional (shaded) types appearing in the type hierarchy in Figure 2. These additional types are only available in SMART 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 structs, 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

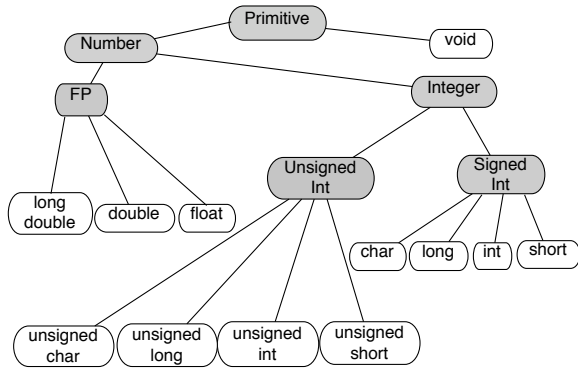


Figure 2. Primitive type hierarchy used for matching in SMART C.

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 calls to both `malloc` and `calloc`. Pattern names can also represent literals by using quotes. The literals matched will depend 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 pattern primitives for all arithmetic, comparison, and logical operators, in addition to other primitive operations of the C language (dereference, address-of, array access, struct 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.

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 are 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 the match site. The modifier may take on the values of `before`, `after`, or `around` (terms borrowed from aspect-oriented programming [14]), 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* allows for the body to be customized based on these properties.

Context variables. Context variables are placeholders 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 is 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 the expression which matches the pattern can be accessed by `tc_expr`. Each operand can, in general, be an arbitrary expression with some

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

Table 1. Context Variables.

type. The type and value of the expression that are matched on are accessed through the `tc_type` and `tc_operand` context variables. For unary operations, there will be one set of type and expression context variables (`tc_type`, `tc_operand`), and binary operations will have two sets (`tc_type`, `tc_operand`, `tc_type2`, `tc_operand2`). The operation is accessed by calling 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 denoted by `tc_args`. This is an array of arguments, each possessing type and expression information. For example, the context variable `tc_args[1].expr` is a placeholder for the value 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 through 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. 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 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 [11] 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 [11]) 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-

```

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.

unaware. This function call is transformed to an appropriate type-dependent `printf()` call.

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 will 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. Once a transformation specification has been defined, SMART C inserts all variables and functions that were declared at the top level of the transformation specification. Then, SMART C must perform pattern matching to determine where to apply the s-macros. First, the SMART C pattern matcher walks through the list of declarations in the program, attempting to match each against any of declaration patterns, if present. 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 corresponding to the source code 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 operates independently of all other transformation specifications. Multiple transformation specifications can then be composed, with each transformation specification being applied in full before the next may begin. This allows transformation specifications to be modular, as is unaware of the operation of any other transformation specifications. For example, a programmer could write a transformation specification that added null pointer checks, as well as one that logged all function calls. These could be written separately, and applied in either order, with the according semantics.

For each transformation specification, each source code primitive may match and be transformed by at most one s-macro. This requires that some ordering be imposed upon the s-macros, and that only the first match of some source

code primitive be transformed. SMART C processes the s-macros in the order the programmer has arranged them in the source file. This gives the programmer flexibility to determine how s-macros should be applied.

3.2. Code Transformation

Once SMART C has found a pattern match, it must determine the transformation to apply to the match site. SMART C must resolve both the transformation-time control constructs and context variables that appear in the transformation body. All of the context variables may be determined statically 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. An expression may be transformed by a s-macro using the `before`, `after`, or `around` 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 contain 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 its associated scope, and may replace or add to the declaration and the associated code. 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 similar. An example of changing a variable declaration has already been presented in Figure 1(b). Examples demonstrating an-

other 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 [16]. 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 specification are dismantled using C-Breeze's built-in dismantler, resulting in a three-address-code-like intermediate representation. Then, we parse the transformation specification(s). The next phase walks through the dismantled AST to search for C constructs which match a 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 is a predefined phase which "undismantles" the code and converts 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.

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.

<p>Source code:</p> <pre>int *ptr, result; ptr = (int *) malloc(sizeof(int)); result = *ptr;</pre> <p>S-macro:</p> <pre>before(*Any* %) { if (tc_operand == 0) error("NULL DEREF") }</pre>	\implies	<p>Transformed code:</p> <pre>int *ptr, *temp, result; ptr = (int *) malloc(sizeof(int)); if (ptr == 0) error("NULL DEREF"); temp = ptr; result = *temp;</pre>	<p>Source code:</p> <pre>int a, b; a = b * 2;</pre> <p>S-macro:</p> <pre>around(Integer % * "2"){ return tc_operand << 1; }</pre>	\implies	<p>Transformed code:</p> <pre>int a, b; a = b << 1;</pre>
---	------------	--	---	------------	---

(a) Before transformation on dereference expression.

(b) Around transformation on multiply expression.

Figure 5. Example SMART C expression transformations.

<p>Source code:</p> <pre>if (x > 0) foo(x);</pre> <p>S-macro:</p> <pre>before(if) { printf("Test %s", </pre>	\implies	<p>Transformed code:</p> <pre>printf("Test %s", "x > 0"); if (x > 0) foo(x);</pre>	<p>Source code:</p> <pre>for(i=0; i < 10; i++) {printf("i=%d", i); }</pre> <p>S-macro:</p> <pre>around(for){ tc_expr; while(tc_expr2) { tc_block; tc_expr3; }</pre>	\implies	<p>Transformed code:</p> <pre>i = 0; while (i < 10) { printf("i=%d", i); i++; }</pre>
---	------------	--	--	------------	--

(a) Before transformation on if statement.

(b) Around transformation on for statement.

Figure 6. Example SMART C statement transformations.

<p>Source code:</p> <pre>int a; foo();</pre> <p>S-macro:</p> <pre>around(decl(Integer %)) { tc_decl; add_object(tc_name); tc_body; del_object(tc_name); }</pre>	\implies	<p>Transformed code:</p> <pre>int a; add_object(a); foo(); del_object(a);</pre>	<p>Source code:</p> <pre>foo(int a, float b) { do_stuff(); }</pre> <p>S-macro:</p> <pre>around(decl(Any %(...))){ tc_decl { init_table(); tc_body; delete_table(); } }</pre>	\implies	<p>Transformed code:</p> <pre>foo(int a, float b) { init_table(); do_stuff(); delete_table(); }</pre>
---	------------	---	--	------------	---

(a) Around transformation on variable declaration.

(b) Around transformation on function declaration.

Figure 7. Example SMART C declaration transformations.

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 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.

The second buffer overflow detection mechanism realized with SMART C is a technique proposed by Ruwase and Lam [20] named 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. The only remaining set of transformations to make is to update the tables appropriately and to perform checks upon pointer dereference. The transformation for binary operations where the left-hand side 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 implemented 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 its `%` directives to write an arbitrary value to memory. Format 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`.

```
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.

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

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

```
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.

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 [21]. 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

A useful tool in profiling applications is a call graph. Call graphs encode the control flow between functions by counting how many times and from where each procedure gets called dynamically. A call graph must be built by observing the program run over some data set. Then, the function calls present in this instance of the executed program are stored somewhere and analyzed afterward.

This is a simple application to implement using 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* [11] 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.

Transformation systems may also operate on keywords introduced by the programmer that will expand into C code. *ASTEC* [18] 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 *MS²* (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) [14] is a general framework for expressing crosscutting concerns in a modular fashion. The most well-known versions are *AspectJ* [13] and *AspectC++* [23]. 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. The transformations are performed at compile-time in the form of a source-to-source transformation. Many systems have brought AOP concepts to C, including *AspectC* [5], *c4* [27], *Aspicere* [1], *Arachne* [10] 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 ex-

pressed 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 aim to provide a machine-specific transformation capability, at the loss of semantic information (such as types), potential optimization opportunities, and portability across architectures. *ATOM* [24] and *EEL* [15] 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* [17], 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 [22] and the Scheme language [12]. Metaprogramming allows programmers to write code which produces other code. Many metaprogramming systems, such as *tcc* [19], 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 and Future Work

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.

Although our experience with SMART C has been very positive and we've found it to be quite powerful despite its simplicity, we believe its generality can be further enhanced

without unduly compromising ease of use. In particular, we intend to introduce limited context sensitivity so that the pattern matching and transformation process may be modified based on the enclosing control structures or properties of the previously transformed code or declarations. For example, this would allow us to transform variable references based upon how the declarations of these variables are transformed. In addition, we intend to explore multi-stage transformations and user-provided annotations for communicating between them. This would allow complex transformations to be broken into components (phases, each of which transforms the code in some manner) so that the transformation of later phases is guided in part by annotations left on the code by earlier phases. Finally, we will pursue the use of limited data-flow properties in the matching and replacement process.

References

- [1] B. Adams and T. Tourwé. Aspect Orientation for C: Express yourself. In *Proceedings of the 3rd 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 the ACM SIGPLAN 1994 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 the ACM SIGPLAN 2000 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 HPCN Europe 1999, the 7th International Conference on High-Performance Computing and Networking (Lecture notes in computer science 1593)*, 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 the 30th Annual 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 the 10th 2001 USENIX Security Symposium*, 2001.
- [8] R. F. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, 1997.
- [9] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. DELI: A New Run-Time Control Point. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 257–268, 2002.
- [10] R. Douence, T. Fritz, N. Lorient, J.-M. Menaud, M. Segura-Devillechaise, and M. Sudholt. An expressive aspect language for system applications with Arachne. In *Proceedings of the 4th International Conference on Aspect-oriented software development*, 2005.
- [11] D. R. Engler. Incorporating application semantics and control into compilation. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, 1997.
- [12] R. Kelsey, W. Clinger, and J. R. (Editors). Revised⁵ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

- [13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*, 2001.
- [14] 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 the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
- [15] J. R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 291–300, 1995.
- [16] 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.
- [17] 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 the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, 2005.
- [18] B. McCloskey and E. Brewer. ASTEC: A New Approach to Refactoring C. In *Proceedings of European Software Engineering Conference/Foundations of Software Engineering*, 2005.
- [19] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, pages 109–121, 1997.
- [20] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the Network and Distributed System Security Symposium*, pages 159–169, 2004.
- [21] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th 2001 USENIX Security Symposium*, 2001.
- [22] T. Sheard and S. P. Jones. Template Meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pages 1–16, 2002.
- [23] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems*, 2002.
- [24] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 196–205, 1994.
- [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. Springer-Verlag, 2004.
- [26] D. Weise and R. Crew. Programmable Syntax Macros. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 156–165, 1993.
- [27] M. Yuen, M. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. Making extensibility of system software practical with the C4 toolkit. In *AOSD Workshop on Software Wngineering Properties of Languages and Aspect Technologies*, 2006.
- [28] C. Zhang and H.-A. Jacobssen. TinyC²: Towards Building a Dynamic Weaving Aspect Language for C. In *Foundations of Aspect-Oriented Languages*, 2003.