
© Copyright 2010 IEEE Computer Society
Automatic Generation of Assertions to Detect Potential Security Vulnerabilities in C Programs That Use Union and Pointer Types

Shamsul Kamal Ahmad Khalid, Jacob Zimmermann, Diane Corney, Colin Fidge
Queensland University of Technology
Brisbane, Australia
s.ahmadkhalid@student.qut.edu.au,
{j.zimmermann, d.corney, c.fidge}@qut.edu.au

Abstract— Type unions, pointer variables and function pointers are a long standing source of subtle security bugs in C program code. Their use can lead to hard-to-diagnose crashes or exploitable vulnerabilities that allow an attacker to obtain privileged access over classified data. This paper describes an automatable framework for detecting such weaknesses in C programs statically, where possible, and for generating assertions that will detect them dynamically, in other cases. Exclusively based on analysis of the source code, it identifies required assertions using a type inference system supported by a custom made symbol table. In our preliminary findings, our type system was able to infer the correct type of unions in different scopes, without manual code annotations or rewriting. Whenever an evaluation is not possible or is difficult to resolve, appropriate runtime assertions are formed and inserted into the source code. The approach is demonstrated via a prototype C analysis tool.

Keywords- program analysis, C, polymorphic types, runtime assertions

I. INTRODUCTION

Many non trivial projects written in C use type unions to allow efficient storing of different types of data in the same memory location. With pointer variables, a programmer can flexibly access any memory location in the address space of a program. Similarly, function pointers allow different functions to be called with a pointer. These constructs provide C programmers with a rudimentary form of polymorphism.

However, in a weakly typed language such as C, the compiler does not check on type compatibility of polymorphic variables such as unions and pointer variables. It is left to the responsibility of the programmer to ensure data type compatibility in operations involving polymorphic variables. The result is a weak type system and consequently a greater deal of “flexibility” in introducing subtle bugs in programs [1]. Even programs such as the Linux kernel, written and extensively reviewed by experts, cannot escape such subtle errors. For example, Figure 3 shows a recent vulnerability found in the 2.6.31 release of the Linux kernel involving a union type. This code can be found at fs/hfs/catalog.c and its purpose is to mount an HFS file system.

On line 27 of function hfs_fill_super() in Figure 3(a) a call to the hfs_bnode_read() function will lead to the simple memory copy operation located in file fs/hfs/bnode.c on line 8 in Figure 3(b). There are no type checks on the buffer buf and node, which is a member of the user controlled structure hfs_find_data, as shown in Figure 1.

Consequently, a user could create an HFS image that would return an HFS entry with len greater than the size of the statically, stack-allocated buf variable which is a union defined in Figure 2. Because of this missing check, a buffer overflow can occur during the memcpy() operation that could overwrite memory space beyond the bounds of buffer buf in hfs_bnode_read().

A patch manually inserted into the code has been developed in this case. However, similar kinds of errors may exist in other parts of the kernel, as unions are widely used. Almost 50% of recent CERT security advisories resulted from buffer overflow attacks made possible by code weaknesses of this kind [2]. Hence, there is a demand for tools able to check for potential type and value inconsistencies in security critical programs, in an automatic and sound way.

Runtime checks help not only during testing, but can also appear in the release version of a program. About 250,000 lines of code of Microsoft Office are assertions, representing 1% of the source code [3]. Chalin surveyed a number of software projects to determine the density of assertion statements in source code and reported an average assertion density of 3.27% in proprietary projects, 5.10% in open source projects, and 6.42% in Eiffel projects [4]. Therefore, reliance on runtime assertions to detect faults in large software is a common practice.
However, assertions are not yet fully utilized for security analysis. One reason is that most software developers have little idea of what information should be specified in an assertion [5]. Furthermore, they will have to manually read through the source code, identify locations that need an assertion, construct the assertion and finally insert them in the source code. Previous work advocated automatic detection of faults using assertions [5],[6],[7], but not automatic identification, construction and insertion of these assertions. Although there is work on automatic generation of assertions, these assertions are produced from formal specifications of the program (e.g., written in JML), not solely from the source code [8].

In this paper, we present an approach for automatic generation of type assertions at locations of potential bugs or vulnerabilities in C programs involving union and pointer variables. Unlike previous work, we do not aim at defining a new, safer programming language; our goal is to automate the analysis and instrumentation of existing programs that may use any programming construct allowed by a gcc compiler.

Our main contribution is a framework for automatically generating assertions at appropriate locations of a source program to detect and prevent exploitable vulnerabilities associated with unions, pointer variables and function pointers. Exclusively from the source code, we identify required assertions, and gather information needed to form the assertions using our own type inference system supported by a custom made symbol table. These assertions are then instrumented into the source program.

II. OVERVIEW OF OUR APPROACH

Figure 4 depicts our complete chain of processes for automatic generation of assertions. Using a type inference system and symbolic execution, our prototype parses plain C source as input. During evaluation of type correctness, each program statement is classified into one of three categories: (i) Proven correct; (ii) Proven incorrect; and (iii) Undecidable. Undecidable cases require information which is either not available at compile time or is too difficult to evaluate statically.

We only describe how we generate assertions for a program containing unions, but the principle for pointer variables and function pointers is the same. The basis of our approach is to evaluate each program statement against a set of typing and symbolic execution rules. In canonical form each rule is of the form $\alpha ^ \beta ^ \gamma ^ \ldots$ If one of the rule’s terms is evaluated as Undecidable, we generate an assertion

```
1 */ hfs_read_super()
2 *
3 /* This is the function that is responsible for mounting an
4  * HFS filesystem. It performs all the tasks necessary to
5  * get enough data from the disk to read the root inode.
6  * This includes passing the mount options, dealing with
7  * HFS file allocation bitmap blocks, calling hfs_btree_init() to get
8  * the necessary data about the extents and catalog B-trees
9  * and, finally, reading the root inode into memory.
10 */
```

---

**Figure 3. A type union bug in the Linux Kernel:**
(a) Function `hfs_fill_super();`
(b) Function `hfs_bnode_read()`
To achieve this, we initialize its type with a symbolic type, otherwise we—

The correctnes

We

we need to ensure the supplied union m has a member selection a of type int at runtime. To achieve this, we need to generate an assertion before k=m.a; as shown in Figure 7. Note that gcc allows statements before variable declarations in a body of a function. Due to this assertion, the table will be updated accordingly—the symbolic type of m will now be associated with field int a, as shown in Figure 6(b). Furthermore, due to the same assertion, a previously undecidable case at line 17 in Figure 7 will become a decidable case, by using the information available in the table. Symbolic evaluation of the union on line 17 will trigger a lookup on m, and due to conflicting types between f and m, the type evaluation fails, and the statement is determined to be incorrectly typed statically, even though the gcc compiler fails to detect this.

Based on this analysis our tool generates a “diff” file for patching the unsafe code with runtime assertions. The type_is(m,a) procedure in Figure 7 is a function to check whether the current type of union m is a’s type. The table is updated by first usage or by first definition of the union. A similar approach is used for maintaining information about and evaluating the current types of pointer variables and function pointers.

III. DESIGN AND IMPLEMENTATION

Our prototype evaluator is implemented as an independent program using the sparse 0.4.1 library [9]. The sparse library provides functionality to parse a C source file and produce an abstract syntax tree (AST) for further processing. We traverse the AST and build our own shadow symbol table for tracking and updating polymorphic variables in the source file. The following sections describe how we construct the symbol table, perform a static analysis and finally show the algorithms that we use to generate the assertions.

A. Symbol Table Maintenance

While traversing the AST, if a new variable symbol with a polymorphic type is found, an entry is created in the symbol table as demonstrated in Figure 6(a) for the program in Figure 5. If a pointer variable declaration is made at line 9 for the program in Figure 5, such as void *s;, an entry @s will be added to the symbol table. In general, if the symbol is a union, a pointer variable or function pointer, we add this symbol with a symbolic type, otherwise we add the symbol with its concrete type.

It becomes more difficult when a variable with polymorphic type is embedded inside a compound structure. Without knowing that a structure contains a polymorphic variable, appropriate entries will not be created in the symbol table. For example, when a lookup is made upon a member selection of a union inside a structure variable, the symbol table will have no information about it. Therefore, it will treat the entry as non-existent, resulting in inaccurate evaluation.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Type</th>
<th>Member Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>m @m</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) Figure 6. The state of the symbol table (a) before; and (b) after line 11 in Figure 5 is evaluated

```c
#include <stdio.h>
#include "libassert.h"

typedef union u {
  int a;
  float b;
} uni_u;

typedef union m {
  int n;
  float f;
} uni_u;

int main(void) {
  uni_u m;
  open_st("something");
  add_polyvar("m");
  relate();
  float f;
  int k;
  assert(type_is(m, "a", "int");
  k = n.a;
  f = n.b;
  close_st();
  return k;
}
```

(b) Figure 7. Patching the source with the “diff” file will instrument the original source with runtime assertions

and insert it into the code to evaluate it precisely at runtime; in other words, if it cannot be proven statically that the rule is guaranteed to hold, we seek to prove dynamically that it holds in each particular execution of the program.

We demonstrate the instrumentation process with a simple example of the C program shown in Figure 5. Although the program compiles using the gcc compiler without errors, it actually contains a logical error at line 12. The correctness of member selection on line 11 depends on what is supplied to the union parameter m. But, m is unknown at compile time. We determine this by doing two things. Firstly, we insert all variables found in the function parameter and all local variables into a symbol table. For a union such as m, we initialize its type with a symbolic type, as shown in Figure 6(a). Secondly, each symbolic evaluation of a union is preceded by a lookup to the symbol table. In this case, the symbolic type @m is returned when union m is evaluated on line 12. As it involves a symbolic type, this evaluation is an undecidable case (i.e., not enough information is available statically).
1 #include <stdio.h>
2 typedef struct {
3   int a;
4   struct {
5      char b;
6      void *c;
7      union {
8         int a;
9         union {
10        int a;
11        double c;
12      } b;
13   } d;
14   void *m;
15   } g;
16   void *j;
17  } deep;

Figure 8. An example of unions deeply embedded inside structures

```
typedef struct {
  int a;
  struct {
    char b;
    void *c;
    union {
      int a;
      union {
        int a;
        double c;
      } b;
    } d;
    void *m;
  } g;
  void *j;
} deep;
```

Figure 9. Initial state of the table after the declaration of variables u and t in Figure 8

Our strategy, therefore, is to flatten all compound data within a particular scope. By doing this, we have separate memory locations for all members of a compound variable. As a result, it is easier to track the current state of a member selection/type of a particular polymorphic variable. For example, analyzing the function `something_else()`, in the program in Figure 8 will result in the symbol table shown in Figure 9. This kind of declaration is fairly common in the gtk framework, an object oriented framework in plain C that implements inheritance and polymorphism features.

We flatten each compound variable using the following method: 1) As we parse each member of a compound variable, we create a unique entry into the global symbol table by concatenation of field names of the current member with its parent’s field name; 2) If the member is a union or pointer variable, we assign a symbolic type (resembling the unique id to maintain uniqueness of the symbolic type, see Figure 9), else we assign it its concrete type.

B. Static Analysis (Type Checking)

Once the symbol table has been initialised, all statements and expressions can be analysed, including those involving unions and pointer variables. Type compatibility checking is performed for every assignment, function argument-parameter and function return. As stated in Section 2, wherever possible we analyse type compatibility statically. For example, while the assignment on line 21 of Figure 8 will be type checked as correct, the assignment on line 22 will show a type incompatibility (storing a floating point number into an integer’s memory space). By contrast, as at least one side of the assignment involves a symbolic type, the type compatibility of the statement on line 23 is undecidable. Due to this, an assertion needs to be inserted before it, such as assert(type_is(t.s.d.b, u.s.d.b)) (i.e., @t.s.d.b == @u.s.d.b). We give the assertion generation algorithm in the next section.

C. Assertion Generation

Algorithm 1 is used for static type checking as well as for generating type assertions. Respectively, T and S can be target and source symbols of an assignment, a function’s argument-parameter or a function’s return types. We can see from the algorithm that if the target or source is a polymorphic variable and still has a symbolic type at the end of its analysis, we will generate the appropriate assertion. Assertions for pointer variable and function pointer evaluations are also generated by Algorithm 1.

Algorithm 2 shows assertions generated due to compound member selection. This is only called for union members (not structures, as they have concrete types). In short, for unions an assertion is generated not only for whole union assignment involving symbolic types but also for usage member selection.

```
Input: Target symbol, T and Source symbol, S
Output: True if T and S are compatible, False if not;
An assertion is generated if S or T is still symbolic.
```

```
if T is a polymorphic variable then
  if T’s type is symbolic then
    Generate an assertion that T’s type is equal to S’s type
    Update T’s type with S’s type (either concrete or symbolic)
    return true
  else if S’s type is symbolic then
    Generate an assertion that S’s type is equal to T’s type
    Update S’s type with T’s concrete type
    return true
  endif
else if S’s type is symbolic then
  Generate an assertion that S is equal to T’s concrete type
  Update S’s type with T’s concrete type
  return true
endif
```

return basetype of T == basetype of S

Algorithm 1. Assertion Generation in the Type Compatibility Checking Module
Input: A union with a selected member
Output: if legitimate then return the selected member
else return error

if member selection occurs as a definition then
Update the parent’s entry in the symbol table with this member
else if member selection occurs as a usage */
Lookup the parent’s current member selection in the symbol table
if lookup returns a symbolic type then
Generate an assertion that the parent’s type is equal to this member’s type
else if lookup result is NOT equal to the selected member then
return error
endif
endif
return this member

Algorithm 2. Assertion Generation in the Compound Member Selection Module

D. Implementation Status

About 1100 lines of codes have been added to the original sparse type evaluator. Our assertion library (libassert.h) provides all necessary data structures and functions to dynamically update and check the latest type information. These functions include runtime opening and closing of the function stacks, updating the types, lookup and mechanisms to inform the called function about any polymorphic parameter-argument relationships, if there is any.

The placement and shape of the inform parameter-argument relationship function is currently being optimized to support more complex expressions. We will put these functions within the scope of if/while statements, instead of outside. Currently, we implement the library using simple string comparisons to demonstrate proof-of-concept. We are converting it to offset based comparisons.

IV. CASE STUDY

We return to the motivational example given in Figure 3. Using the approach we have described above, the static analyser will recognize that the union type buffer, buf is not yet defined in the context of the hfs_bnode_read() function. Therefore, Buf will have a symbolic type in this context. Furthermore, knowing that memcpy() is effectively an assignment operation, the analyser can determine if assertions are needed for a secure memcpy() operation. In this case, a type assertion should be inserted right before the memcpy() at line 14, as shown in Figure 10. As explained in Section 2, function type_is(x,y) checks whether the current type of x is compatible with y’s type. Notice that a temporary variable, ‘_temp1’ must be generated to ensure that only a single call is made to the kmap() function, which could return different values if called more than once.

The static analyser could also perform symbolic execution to determine the buffer length of both the memory range in the destination and the source of the call to memcpy(). As both lengths are unknown (i.e., they are symbolic lengths), a runtime assertion can be inserted before memcpy(), as shown in line 13 of Figure 10. Function buflen_gt() will return false if the buffer length of the source is greater than that of the destination. (Generating bounds checks such as that shown in line 13 is straightforward [10] and is not done by our current prototype which focuses on type correctness.)

V. RELATED WORK

To make C safer, new languages have been developed with changes in syntax by putting new restrictions on standard C, and by adding more annotations. The Cyclone language [11], a dialect of C, provides mechanisms such as sum types and subtyping within C, allowing safer programs to be written. Cyclone extends the union type by adding a tag to each case of the union. Arguments with unions in Cyclone are treated as tagged-unions. One of our advantages over Cyclone is that we do not modify existing data structures, i.e., we are fully backward-compatible and interoperable. Functional languages like Haskell and ML provide disjoint sum types within the language, enabling precise typing of compound objects [12]. These new languages are safer but require programs to be redesigned. This is not always possible as developers may need the flexibility of memory access in standard C to achieve performance and compactness. Furthermore, the conversion has to be done manually, and there is no guarantee that the new dialect does not introduce new problems. We instead focus on making legacy code written in C safer.

There has been much recent work on statically proving memory safety of C programs to make them execute safely [6],[11]. However, the latest commercial analysers like the Coverity Dynamic Analysers only analyse Java multithreading programs, disregarding legacy code written in C. Another major player, Klocwork, focuses purely on static analysis techniques. As certain evaluations are not statically possible, some analysers add runtime checks. For example, CCured adds runtime checks for certain premises in its pointer-kind type inference rules [1]. But, for unions, CCured either adds additional tags or removes unions altogether in the program. We instead modify the union structures and may not work for hardware-dependent applications where the data layout cannot be

Figure 10. Program in Figure 3 instrumented with appropriate runtime assertions
changed. Runtime type information has been used for bug finding and providing debugging information for bad casting or union access [6], but these assertions are manually formed and inserted into the source. Avots et al. [13] use dynamic type checking but because the type of a global or dynamically allocated object is taken to be the type of its first dereference, they produce false positive type violations, which they must remove manually. Instead, we rely exclusively on source code to identify, form and insert the runtime assertions, relieving the burden of developers of doing these jobs manually. Unlike Yong and Horwitz [14], we instrument both reads and writes.

Jones and Kelly presented a backward compatible bound checking method and implemented it in gcc [10]. In their implementation, all known valid storage objects are maintained in a table, and one can use the table to map a pointer to a descriptor of the object into which it points, which contains the base, extent and additional information to improve error reporting. They did not, however, support tracking of types in unions.

Xu et al. [15] presented a transformation of C programs to a “safer C” but it may not always be practical for existing applications. They use metadata to track information about pointers and unions but focus more on correctness than vulnerabilities. They use a runtime mark for unions which is nearly identical to ours, but they do so only because they need it in case the union contains a pointer type. They do not use symbolic evaluation; instead, they generate assertions for every reference. They mention static analysis only as a possible optimization technique in the “Global Optimizations” section, but, while they consider it very promising, it is intended for future work. Similarly, Dhurjati and Adve [16] modify function interfaces and function calls to add metadata. They claim to have low overhead compared to other approaches primarily due to the use of Automatic Pool Allocation for memory allocation. However, this changes the underlying architecture significantly. There is no mention of how they treat unions.

Finally, Rosenblum and Clark [5][17] empirically identified categories of assertions that can be written into C programs to discover faults, but not vulnerabilities.

VI. CONCLUSION AND FUTURE WORK

We have explained our approach for analysing C source code programs involving unions and pointer variables that may lead to security vulnerabilities such as buffer overflows. We use static analysis to resolve variable types where possible, but generate runtime assertions when static checking is insufficient.

At the time of writing we have completed a prototype implementation of our approach which performs the necessary static analysis and identifies points in the program where assertions are needed. Using information gained from the static analysis, it forms and generates the assertions into a “diff” file which can be used to produce the secure, instrumented program. In future, we will add more elaborate assertions to handle various forms of C constructs, such as buffer length assertions, loop assertions, interfile assertions, pointer assertions, etc.

ACKNOWLEDGMENTS

We wish to thank the anonymous referees for their helpful comments. This work was supported in part by the Defence Signals Directorate via ARC Linkage Grant LP0776344.

REFERENCES