Testing Inexecutable Conditions on Input Pointers in C Programs with SANTE

Omar Chebaro
ASCOLA (EMN-INRIA, LINA), Ecole des Mines de Nantes
44307 Nantes France
Email: omar.chebaro@mines-nantes.fr

Mickaël Delahaye
UJF-Grenoble 1, LIG, UMR 5217
38041 Grenoble France
Email: mickael.delahaye@imag.fr

Nikolai Kosmatov
CEA, LIST, Software Reliability Laboratory, PC 174
91191 Gif-sur-Yvette France
Email: nikolai.kosmatov@cea.fr

Abstract. Combinations of static and dynamic analysis techniques make it possible to detect the risk of out-of-bounds memory access in C programs and to confirm it on concrete test data. However, this is not directly possible for input arrays/pointers in C functions. This paper presents a specific technique allowing the interpretation and execution of assertions involving the size of an input array (pointer) of a C function. We show how this technique was successfully exploited in the SANTE tool where it allowed potential out-of-bounds access errors to be detected and classified in several real-life programs.
Keywords: run-time errors, C pointers, static analysis, test generation.

1. INTRODUCTION

The C programming language is paradoxically both the most commonly used for development of system software with various critical components, and one of the most poorly equipped with adequate protection mechanisms. The C developer is responsible for correct allocation and deallocation of memory, pointer dereferencing and manipulation (like casts, offsets, etc.), and even for the validity of indices in array access operations. Detecting potential errors in C programs remains one of the most challenging topics in modern software verification.

Among the most recent research advances in this domain, various combinations of static and dynamic analysis tools were shown to be advantageous for verification of C programs. One of them, SANTE (Static ANalysis and TEsting) [5] combines abstract interpretation and test generation. It uses the value analysis plugin [4] of FRAMA-C [7] to detect and report potential runtime errors in a C program, that are then classified (i.e. confirmed or infirmed) by the dynamic symbolic execution (DSE) based test generation tool PATHCRAWLER [3].

A frequent cause of errors in C programs is an invalid pointer or array access. It may result in particularly dangerous faults, such as runtime errors and buffer overflows, often exploited in attacks. Some simple protection mechanisms, e.g. involving the size of input arrays and pointers in C functions, are impossible to introduce or even ignored according to the C norm (cf. Sec. 2). This paper focuses on the problem of detection of this kind of errors by combined static-dynamic analysis tools like SANTE, trying to automatically confirm a detected risk by concrete execution of a test. This problem has not been addressed in previous publications and experiments on SANTE.

Contributions of this paper. We present the problem of expressing and executing assertions to prevent out-of-bounds access for input arrays (pointers) in C functions, that appears in particular when combining static and dynamic analyses (Sec. 2). We describe the solution implemented in the SANTE tool that makes it possible to express the assertions involving the size of an input array (pointer), to interpret and to execute them in a dynamic analysis tool (Sec. 3). A short experience report illustrates how this technique allowed potential out-of-bounds access errors to be detected and classified with SANTE in real-life programs (Sec. 4).

2. INEXECUTABLE ASSERTIONS PREVENT THE CONFIRMATION OF ERRORS

As said earlier, the validity of pointers is very difficult to check dynamically. Indeed, languages with pointers, such as C or C++, do not allow the developer to check for their validity. The developer is supposed to know when a pointer is valid or not, possibly by using well-known conventions. These conventions include the use of a unique special value for invalid pointers, for instance, (void*)0 or NULL. However, such conventions can be difficult to enforce, and do not address the problem of out-of-bounds access.

An existing verification approach in combined static-dynamic analysis tools uses first static analysis to detect a *threat* (i.e. a potentially invalid array access or pointer dereference), and marks it with an annotation specifying the condition that should be met to avoid the error. Then the dynamic analysis step tries to confirm this threat on a concrete test. If the threat is a false alarm, it cannot be confirmed.

```
int a1[10]; //global array
void f1(int i){
   ...
//@ assert i>=0 && i<10;
   a1[i]=0;
   ... }
```

```
void f2(int i){
   int a2[10]; //local array
   ...
//@ assert i>=0 && i<10;
   a2[i]=0;
   ... }
```

Fig. 1. Examples where the array size is known and can be used to express the precise assertion.

```
//input array
void f3(int a3[10],int i){
   ...
//@ assert \valid(a3+i);
   a3[i]=0;
   ... }
```

```
//input pointer
void f4(int *p4,int i){
   ...
//@ assert \valid(p4+i);
   p4[i]=0;
   ... }
```

```
int *p5; //global pointer
void f5(int i){
   ...
//@ assert \valid(p5+i);
   p5[i]=0;
   ... }
```

Fig. 2. Examples where the array size is ignored or unknown. The assertion cannot use it.

Suppose that, in each example of Fig. 1, the array access may be potentially out-of-bounds. In the SANTE tool, the value analysis step detects the threat and inserts an annotation specified in ACSL specification language [2] by the assert keyword. For global or local arrays, the array dimension is known and the generated annotation explicitly gives the condition of error-free behavior. The error condition is easily obtained by negation. It can be both executed and used as a test objective to guide PATHCRAWLER.

Let us now consider the programs of Fig. 2, where each array (pointer) access is supposed to be potentially out-of-bounds. Although the size of the input array a3 is provided in f3, it is ignored according to the C norm [11, Sec. 6.7.5.3.7]. In other words, the declaration int f3(int a3[10], int i) is equivalent to int f3(int *a3, int i), so the array size is lost. At runtime, sizeof(a3) returns the size of a pointer. Nothing guarantees that a3 really refers to an array with 10 elements. In this case, the value analysis step in SANTE generates a general annotation \valid(a3+i) requiring the validity of the array access. It is impossible to be more explicit since we cannot specify the allowed interval of values for the index i. The examples f4 (with an input pointer) and f5 (with a global pointer) have a similar problem. In all these cases, the annotation provides an inexecutable condition and cannot be directly used to guide test generation and to confirm the potential error on some generated test data. Global variables being seen as generalized inputs in unit testing, we group the three cases of Fig. 2 together under the term *input array (pointer).*

3. THE METHOD

This section presents the technique allowing us to express and to interpret, both symbolically and concretely, the number of elements referred by an input array (pointer). We will represent this number using a special function pc_length. Inexecutable assertions that needed this size can now be expressed, executed and tested in a DSE testing tool like PATHCRAWLER. Our technique contains three distinct parts: first, threats have to be translated into program statements; second, the program launcher is modified accordingly to allow dynamic checking of array bounds involving pc_length; third, the same statements are given a particular symbolic interpretation, allowing the testing tool to search for test inputs violating the assertion. The three following subsections detail these three parts.

Fig. 3 shows our running example. This program compares the first n numbers in the two given arrays of integers with respect to the lexicographic order. At line 5, the program accesses t1[i] and t2[i]. We assume that static analysis detects a threat for both expressions and inserts the corresponding assertion at line 4.

```
1       void tuplecmp(size_t  n, const int* t1, const int* t2) {
2         size_t  i;
3         for (i = 0; i < n; i++) {
4           /*@ assert \valid(t1+i) && \valid(t2+i) */
5           if (t1[i] > t2[i]) return 1;
6           else if (t1[i] < t2[i]) return -1;
7         }
8         return 0;
10      }
```

Fig. 3. Function tuplecmp compares the first n elements of given arrays t1, t2

3.1. From threats to statements

The first step consists in translating an inexecutable ACSL annotation assert \valid(p+j) for an input array (pointer) p into a C statement using a function call pc_length(p) supposed to return the size of p. The annotation assert \valid(p+j) is translated as

```
if ( j < 0 || j >= pc_length(p) ) {
  pc_assertion_error();
}
```

If the error condition is true, the function pc_assertion_error reports the error and exits; otherwise the execution continues normally. This new decision (branch) with the explicit error condition adds a test objective in PATHCRAWLER so that test generation will try to confirm the threat. For example, the line 4 in Fig. 3 is replaced by

```
if ( i < 0 || i >= pc_length(t1) || i >= pc_length(t2) ) {
  pc_assertion_error();
}
```

3.2. Backing the concrete execution of pc_length

To obtain meaningful information from concrete execution, PATHCRAWLER combines a specifically instrumented version of the program under test and a program launcher. The instrumentation consists in injecting special statements to track the execution flow in the program. The program launcher is in charge of initializing communication streams and executing the instrumented function under test on each test. While the instrumentation does not require any modification, the memory initialization has to be modified.

Memory initialization. Fig. 4 shows (a simplified version of) the launcher generated by PATHCRAWLER for the function of Fig. 3. The launcher consists of a loop that allows PATHCRAWLER to execute multiple tests in a single run of the launcher. At each iteration, the launcher receives from the generator the input values of a test and calls the function under test on these values. For non-pointer parameters, values are transmitted directly to the function as actual parameters. However, for input arrays (pointers), sufficient memory must be allocated. That is why the launcher first expects the size of the array to allocate. Then, after allocating required memory space with pc_array_alloc, it reads a value for each cell of this memory block. Fig. 5 illustrates the values sent to the launcher for a test and their meaning, as well as the corresponding function call.

Keeping track of length. The key modification allowing pc_length(p) to obtain the size of an input array (pointer) p during concrete execution is made in the function pc_array_alloc. This function shall now keep track of the memory size for each pointer allocated. We introduce a global dictionary D recording each pointer allocated with its size. Each call p=pc_array_alloc(l,e) for an array of l elements of size e first allocates the required memory space (l×e bytes) for p using a standard malloc and records p with its size in D. The function pc_length(p) has simply to query the dictionary D in order to find the number of elements associated with p.

```
1   int main () {
2       int n; // scalar parameter
3       int *t1 , *t2; // array parameters
4       int _len, _i; // launcher variables
5       pc_init_streams ();
6       while (! pc_check_if_finished ()){
7           pc_scan(INT, &n); // receiving test data
8           pc_scan(INT, &_len);
9           if (_len == 0) t1 = pc_null ();
10          else {
11              t1 = pc_array_alloc(_len, sizeof(int));
12              for (_i = 0; i < _len ; i++)
13                  pc_scan(INT, &t[_i]);
14          }
15          pc_scan(INT , &_len);
16          if (_len ==0) t2 = pc_null ();
17          else {
18              t2 = pc_array_alloc(_len, sizeof(int));
19              for (i = 0; i < len ; i++)
20                  pc_scan(INT, &t2[_i]);
21          }
22          tuplecmp(n, t1, t2); // calling function under test
23          pc_call_oracle (); // calling the oracle
24          pc_free(t1); // deallocating memory
25          pc_free(t2);
26      }
27  }
```

Fig. 4. Generated launcher for the function tuplecmp of Fig. 3

Test data received by the launcher | **Function call**

Value	Meaning
3	n
2	size of t1
0	t1[0]
1	t1[1]
3	size of t2
5	t2[0]
-2	t2[1]
7	t2[2]

→

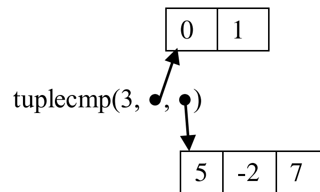

Fig. 5. Launching the function tuplecmp of Fig. 3 on a test with input arrays

3.3. Backing the symbolic execution of pc_length

Arrays and pointers are known to be hard to handle in constraint-based test generation [12, 17]. In forward symbolic execution, used in most implementations of DSE, it is usually possible to transform input pointer dereferences into indexed array accesses. Consequently, if we can handle arrays in constraints, input pointers can be for the most part handled as arrays. However, conditions on array lengths are still challenging. Indeed, these conditions require that array lengths are modeled, which may sometimes not be the case like in the theory of arrays with extensionality [13, 16].

Fortunately, a frequently adopted solution (used in PATHCRAWLER) is to associate each array a with a logicial variable len_a representing its length. Thanks to the length variables, symbolic execution of calls to pc_length is straightforward: pc_length(a) is simply replaced with len_a and its implementation is ignored. For instance, the condition i >= pc_length(t1) can be translated by the constraint $i >= len_{t1}$.

4. EXPERIMENTS WITH SANTE

This section provides a short experience report illustrating how the technique presented in Sec. 3 allows the SANTE tool to treat threats involving input arrays (pointers). Our recent experiments on several real-life programs show that such threats are very widespread. Fig. 6 presents five examples. Its columns provide the example number, module and function name, its size in lines of code, the number of threats reported by value analysis, the number of threats involving input arrays (pointers) and the number of known bugs. It shows that out of 41 reported threats, 37 (90%) involve input arrays (pointers). For Ex. 2, 3 and 4, all reported threats are out-of-bounds accesses in input arrays or pointers. All bugs are related to input arrays (pointers).

№	*(module /) function*	*size (LOC)*	*reported threats*	*threats involving pointers*	*bugs*
1	libgd / gdImageString-FTEx	705	12	10	1
2	Apache / get_tag	696	12	12	3
3	polygon	202	10	10	2
4	rawcaudio / adpcm_decoder	365	2	2	0
5	eurocheck	154	5	3	1
Total			41	37	7

Fig. 6. Number of threats, threats involving pointers and bugs in our experiments

Let us illustrate the SANTE results on the open-source program[1] eurocheck. This program validates serial numbers of European bank notes. It takes an input string str representing the serial number. str can be NULL or a zero-terminated string with variable length. The value analysis step in SANTE reports 5 threats of potential runtime errors. In Fig. 7a, we give a simplified version of the program with the assertions added by the value analysis at lines 6_0, 13_0, 15_0, 17_0 and 20_0. Fig. 7b presents the program after the translation of ACSL annotations into C statements.

The assertion at line 17_0 reports that writing in checksum[i] might be an out-of-bounds access. In this case, the array bounds are known since checksum is a local array. The error condition (line 17_1) is directly obtained by negating the error-free condition given by the assertion. The threat at line 20 is treated in the same way.

The situation is different for threats at lines 6,13,15 related to the input pointer str. At line 6, we read the first character without verifying if the string is NULL or not. The reported threat indicates that str+0 may be invalid. The dynamic analysis step will try to generate a test where str+0 is invalid. Here we need the array size to express the error condition (line 6_1). In this case, PATHCRAWLER is able to generate a test case where str = NULL violating this assertion. For 13_0 and 15_0, the error condition added by SANTE is i<0 || i>=pc_length(str). PATHCRAWLER detects that all paths violating these assertions are infeasible and, therefore, these threats are false alarms.

[5] gives other experimental results and compares the results of SANTE to static analysis and test generation used separately. Notice that, over 41 threats, only 6 remain unclassified, i.e. SANTE cannot determine whether a threat is a real bug or a false alarm. SANTE appeared to be in average 43% faster than test generation alone. The number of remaining unclassified threats with SANTE decreases by 82% with respect to test generation alone, and by 86% with respect to value analysis alone.

5. RELATED WORK AND CONCLUSION

We presented an original solution filling the gap between a static analysis tool, being able to report (by an inexecutable annotation) a potential out-of-bounds access operation in input arrays (pointers) in C functions, and a DSE testing tool, requiring a more precise and executable assertion in order to guide test generation and to confirm the erroneous behavior. We illustrated how the combined tool SANTE essentially relies on this solution that efficiently serves for a very frequent type of threats.

In the security context, out-of-bounds array accesses have been previously investigated in C, notably to prevent buffer overflow exploits. For instance, some authors propose to secure each and every array or pointer access by adding dynamic checks using either a dedicated C compiler [1], or specific program

[1] http://freshmeat.net/projects/eurocheck

transformations [8, 14]. Others like Necula et al. [15] propose to use static analysis to detect safe pointer access to skip some dynamic checks. As in these later approaches, our approach does not consider every pointer access, but allows us to treat only those threats that are not discarded by static analysis. However, the use of a general-purpose specification language like ACSL allows much more choice in the kind of static analysis being used. In addition, our approach uses dynamic symbolic execution to classify the detected threats.

```
0    int eurocheck( char *str ){
1      unsigned char sum;
2      char c[9][3] = { "ZQ", "YP", "XO",
3        "WN", "VM", "UL", "TK", "SJ", "RI"};
4      unsigned char checksum[12] ;
5      int i = 0, len = 0;
6_0    //@assert \valid( str+0 );

6      if( str[0]>=97 && str[0]<=122)
7        str[0]−=32;
8      if( str[0]< 'I' || str [0]> 'Z' )
9        return 2;
10     if( strlen( str ) != 12)
11       return 3;
12     len = strlen( str );
13_0   //@assert \valid( str+i );

13     checksum[i] = str[i] ;
14     for ( i =1; i<len ; i++){
15_0     //@assert \valid( str+i );

15       if ( str[i]<48 | | str[i]>57)
16         return 4;
17_0     //@assert 0<=i && i<12;

17       checksum[i] = str[i]−48;}
18     sum=0;
19     for( i=(len -1); i>=1; i--)
20_0     //@assert 0<=i && i<12;

20     sum+=checksum[ i ]; }
21     while( sum>9)
22       sum = ( ( sum/10 ) + ( sum%10) ) ;
23     for( i=0; i<9; i++)
24       if ( checksum [0] == c[i][0]
25         || checksum [0] == c[i][1] )
26         break ;
27     if( sum != i )
28       return 5;
29     return 0;}
```

a) Assertions generated by value analysis

```
0    int eurocheck( char *str ){
1      unsigned char sum;
2      char c[9][3] = { "ZQ", "YP", "XO",
3        "WN", "VM", "UL", "TK", "SJ", "RI"};
4      unsigned char checksum[12] ;
5      int i = 0, len = 0;
6_1    if(0<0 || 0 >= pc_length( str ) ){
6_2      pc_assertion_error(); }
6      if( str[0]>=97 && str[0]<=122)
7        str[0]−=32;
8      if( str[0]< 'I' || str [0]> 'Z' )
9        return 2;
10     if( strlen( str ) != 12)
11       return 3;
12     len = strlen( str );
13_1   if( i<0 || i >= pc_length( str ) ){
13_2     pc_assertion_error(); }
13     checksum[i] = str[i] ;
14     for ( i =1; i<len ; i++){
15_1     if( i<0 || i >= pc_length( str ) ){
15_2       pc_assertion_error(); }
15       if ( str[i]<48 | | str[i]>57)
16         return 4;
17_1     if (!(0<= i && i <12)){
17_2       pc_assertion_error(); }
17       checksum[i] = str[i]−48;}
18     sum=0;
19     for( i=(len -1); i>=1; i--)
20_1     if (!(0<= i && i <12)){
20_2       pc_assertion_error(); }
20     sum+=checksum[ i ]; }
21     while( sum>9)
22       sum = ( ( sum/10 ) + ( sum%10) ) ;
23     for( i=0; i<9; i++)
24       if ( checksum [0] == c[i][0]
25         || checksum [0] == c[i][1] )
26         break ;
27     if( sum != i )
28       return 5;
29     return 0;}
```

b) After translation of assertions into C

Fig. 7. Simplified version of eurocheck before and after assertion translation

In code-based test generation, handling pointers is still considered as a challenge. Much research has been done to model pointer accesses. For instance, Elkarablieh et al. [9] propose a very precise modelization of pointer operations. Another approach, proposed by Xu et al. [18], is to abstract such operations to keep only the most relevant operations to detect buffer overflows. Our approach relies on a classical memory model, but it exposes symbolic array lengths to permit the symbolic execution of assertions involving array sizes. This complements the specific concrete execution and allows us to benefit from an efficient DSE test generation.

In existing specification languages, dynamic checking of assertions related to pointer validity is usually impossible. Indeed, since Java does not have pointers and allows array bound checking, such conditions do not

occur in JML [6]. In .NET, if code contracts allow specifying behaviors of "unsafe code" (a mode without memory management), conditions related to pointer validity are not executable [10]. To the best of our knowledge, SANTE is the only tool using such a translation of annotations for input arrays (pointers) from a specification language into executable code and applying DSE to confirm or infirm the threats.

REFERENCES

[1] Austin, T.M., Breach, S.E., Sohi, G.S.: Efficient detection of all pointer and array access errors. SIGPLAN Not. **29**, 1994

[2] Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. (February 2011) http://frama-c.cea.fr/acsl.html

[3] Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: AST'09

[4] Canet, G., Cuoq, P., Monate, B.: A value analysis for C programs. In: SCAM'09

[5] Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: SAC'12

[6] Cheon, Y.: A Runtime Assertion Checker for the Java Modeling Language. Department of Computer Science, Iowa State University, 2003

[7] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B. Frama-C: A Program Analysis Perspective. In SEFM 2012. http://frama-c.com

[8] Dahn, C., Mancoridis, S.: Using program transformation to secure C programs against buffer overflows. In: WCRE'03

[9] Elkarablieh, B., Godefroid, P., Levin, M.Y.: Precise pointer reasoning for dynamic test generation. In: ISSTA'09

[10] Ferrara, P., Logozzo, F., Fähndrich, M.: Safer unsafe code for .NET. In: OOPSLA'08

[11] ISO/IEC 9899:1999: Programming languages – C

[12] Kosmatov, N.: All-paths test generation for programs with internal aliases. In: ISSRE'08

[13] Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. 2008

[14] Lee, T.R., Chiu, K.C., Chang, D.W.: A lightweight buffer overflow protection mechanism with failure-oblivious capability. In: IA3PP'09

[15] Necula, G.C., McPeak, S., Weimer, W.: CCured: type-safe retrofitting of legacy code. In: POPL'02

[16] Stump, A., Barrett, C.W., Dill, D.L.: A decision procedure for an extensional theory of arrays. In: LICS'01

[17] Visvanathan, S., Gupta, N.: Generating test data for functions with pointer inputs. In: ASE'02

[18] Xu, R.G., Godefroid, P., Majumdar, R.: Testing for buffer overflows with length abstraction. In: ISSTA'08