Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Filtering false alarms of buffer overflow analysis using SMT solvers

Youil Kim [a], Jooyong Lee [b], Hwansoo Han [c,*], Kwang-Moo Choe [a]

[a] *Department of Computer Science, KAIST, Republic of Korea*
[b] *Department of Computing and Information Sciences, Kansas State University, United States*
[c] *Department of Computer Engineering, Sungkyunkwan University, Republic of Korea*

ARTICLE INFO

Article history:
Received 3 April 2009
Received in revised form 30 September 2009
Accepted 13 October 2009
Available online 21 October 2009

Keywords:
Buffer overflow
Program analysis
False alarm
SMT solver

ABSTRACT

Buffer overflow detection using static analysis can provide a powerful tool for software programmers to find difficult bugs in C programs. Sound static analysis based on abstract interpretation, however, often suffers from false alarm problem. Although more precise abstraction can reduce the number of the false alarms in general, the cost to perform such analysis is often too high to be practical for large software. On the other hand, less precise abstraction is likely to be scalable in exchange for the increased false alarms. In order to attain both precision and scalability, we present a method that first applies less precise abstraction to find buffer overflow alarms fast, and selectively applies a more precise analysis only to the limited areas of code around the potential false alarms. In an attempt to develop the precise analysis of alarm filtering for large C programs, we perform a symbolic execution over the potential alarms found in the previous analysis, which is based on the abstract interpretation. Taking advantage of a state-of-art SMT solver, our precise analysis efficiently filters out a substantial number of false alarms. Our experiment with the test cases from three open source programs shows that our filtering method can reduce about 68% of false alarms on average.

1. Introduction

When a program accesses a memory address which is beyond the legal limits of data, we call this access a buffer overflow. Array type data and dynamically allocated data have their own bounds, but programming languages such as C do not check the boundary trespasses for the performance reason. The responsibility to write a correct code is on programmers for such programming languages. In C programs, however, pointer arithmetic operations often accidentally allow programs to access the content of memory beyond the boundaries without any immediate exceptions. Those illegal accesses could alter the important contents near the buffers and even the return addresses. Consequently, buffer overflows often lead to many software bugs and serious security holes. Some of the bugs remain undetected even after extensive tests with many dynamic debugging tools. According to the NIST ICAT vulnerability database, buffer overflow defects account for roughly a third of all the severe remotely exploitable vulnerabilities [20].

In detection of buffer overruns, static analysis tools have several advantages over dynamic detection tools. In dynamic tools, we often need to instrument programs with buffer-overrun checks and run them with test inputs. Since the test inputs rarely represent all the possible execution scenarios we may encounter in real environments, some of the execution scenarios could remain untested. In contrast, static tools are relatively easy to test almost all the execution scenarios. Since they require no real execution with real inputs, sound static tools can even take account of all the possible execution paths with all the possible inputs. Static tools, however, could report too many false alarms due to their overly aggressive abstractions.

In this work, we develop a scalable and yet precise static analysis for the buffer overflow detection in C programs. We design a sound analyzer, which can catch all the possible buffer overflow defects. At the same time, we make our analyzer precise enough to filter out a large portion of the false alarms. The main technique we use is to change the abstraction level only within partial regions of code and filter out the false alarms.

Several works already investigate the static analysis for detecting buffer overflow bugs. Existing works such as BOON [17], Splint [8] and ARCHER [19] can detect buffer overflow relatively fast, but they use unsound techniques, whereas our analyzer uses a sound analysis. In addition, our analyzer requires no annotation of source code, whereas Splint and CSSV [6] require annotations from users.

* Corresponding author. Address: Dept. of Computer Engineering, Sungkyunkwan University, 300 Cheoncheon-dong, Jangan-gu, Suwon 440-746, Republic of Korea. Tel.: +82 31 299 4594; fax: +82 31 299 4921.
E-mail addresses: youil.kim@arcs.kaist.ac.kr (Y. Kim), jlee@cis.ksu.edu (J. Lee), hhan@skku.edu (H. Han), choe@kaist.ac.kr (K.-M. Choe).

0950-5849/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2009.10.004

The C Global Surveyor from NASA [16] is an example of efficient sound analyzers but this is heavily tuned to a particular family of NASA software. Since our target applications are open source C applications, we take a general approach to handle quite a large spectrum of applications.

The base analyzer of our buffer overflow detector is based on the abstract interpretation [3], which guarantees that our analyzer can provide a sound analysis framework. Due to the nature of the abstract interpretation, we frequently need to struggle with a large number of false alarms, which often outnumbers real bugs. Reducing the number of false alarms has been one of the highly interesting tasks among the abstract interpretation community. A traditional approach to reduce the number of false alarms is to employ more precise abstract domains. For example, the abstraction with convex polyhedral domains generally leads to less false alarms than with interval domain. Although more precise analysis could reduce the number of false alarms, the time spent on such analysis tends to increase instead.

In order to reduce the false alarms of our base analyzer, we take advantage of a state-of-the-art SMT solver and filter out improbable alarms. Each access to a buffer, whether it is accessed through an element of an array or a pointer, is checked if the access occurs within the buffer bound. Such checks are usually described as linear arithmetic expressions on the integer domain. Since recent SMT solvers such as Yices [2] are quite good at solving linear arithmetic problems, we translate the alarm statement and its related code into the SMT formulae and solve the linear arithmetic formulae to find out the alarm is improbable. If the alarm condition is proved to be improbable, we can filter out the alarm.

This paper is organized as follows. First, we present the high level overview on our filtering method and describe the idea using an example code. Next, we describe how we use the SMT solver for filtering purpose along with the complete set of translation rules for the SMT formulae. Then, we present our experimental results on test benchmarks. Finally, we discuss related works and conclude our paper.

2. Overview of filtering false alarms

After investigating several open source applications, we notice that most parts of the code can be verified with relatively imprecise analysis and only small fragments of the code actually need expensive but precise analysis. To attain both scalability and accuracy in static analysis, our approach is that we apply precise analysis techniques only to the limited parts of the code where those techniques are beneficial, while we employ aggressive abstractions for the rest of the code.

According to the key idea, we designed a two phase buffer-overflow analysis. The high level structure of the whole system is shown in Fig. 1. In order to detect buffer-overflow defects effectively, we begin with a cheaper and imprecise analysis based on abstract interpretation. As a result, we often get many false alarms from the first analysis. Then, we apply a precise symbolic execution technique based on a state-of-art SMT solver to small areas of the code around the alarms where we can verify if those alarms are false positives.

2.1. The first phase: buffer overflow analysis

The components within the buffer overflow analyzer shown in Fig. 1 represent three analyses that Raccoon [11], our base analyzer, performs for the buffer overflow detection. First, we use a unification-based flow-insensitive and context-insensitive points-to analysis [5] to efficiently resolve pointers to scalar variables or functions. Then, the interval analysis [3] based on abstract interpretation analyzes the possible values for an integer variable and represent them as a pair of minimum value and maximum value. Finally, we compute the detailed information including the buffer bounds of pointer variables. We separately perform three dedicated analyses one by one, rather than analyzing all the information at the same time. We believe such separate execution will be advantageous to minimize the footprint of memory usage, while we keep the precision loss to a minimum. We developed Raccoon to perform a fast point-to and interval analysis for C programs. It has been implemented in Objective Caml by extending the CIL compiler infrastructure [14], which handles the full set of ANSI C syntax with GNU C Compiler and Microsoft Visual C extensions. Our experiments with several open source applications show that Raccoon can finish the three analyses within a hundred of seconds for code with a ten thousand lines. Moreover, it can prove about half of the array and pointer accesses are actually free of buffer overflow (i.e., safe).

2.2. False alarms of Raccoon

Racoon can prove the safety for a large number of buffer accesses, but still produces many false alarms due to the aggressive abstraction in its buffer overflow analysis. One of the primary reasons why Racoon produces many false alarms is the way it handles loop statements. Most static analyzers based on abstract interpretation often exploit widening and narrowing operations [3] for the acceleration and termination guarantee of their analyses. The narrowing operations, however, often fail to recover the imprecision introduced by the widening operations. Relational analysis could help alleviate such inaccuracies that are originated by widening operations in loop statements, but even a powerful polyhedral analysis [4] is incapable of handling all the complex loop statements.

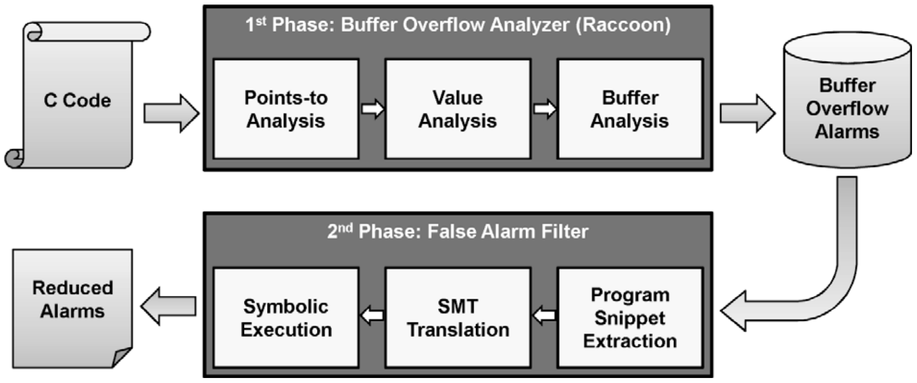

Fig. 1. Two phase buffer overflow analysis: buffer overflow analyzer and false alarm filter. The first phase is composed of three analyses for our buffer overflow analyzer, Raccoon [11]. The second phase performs alarm filtering with three steps.

Another primary reason for false alarms is the way Raccoon performs its analysis. Separate execution of each analysis can help reduce the memory usage, but the information to be generated from the next analysis cannot be used in the current analysis. During the value analysis, for example, Raccoon cannot exploit the information about pointers and maximum lengths of strings. We believe that the separate execution of analyses contributes to the reduced analysis time mainly due to the small memory usage, but it may cause more false alarms.

2.3. The second phase: filtering false alarms

Since false alarms are inevitable particularly in fast static analyses, reducing false alarms by all means is necessary to make static analyses useful. To detect false alarms, we design the second phase of our buffer overflow analysis. By performing a precise symbolic execution over the limited areas of code around alarms, we can filter out a substantial number of false alarms.

In the following algorithm sketch, we describe how our false alarm filtering works. For each alarm statement, we perform the following steps to examine the alarm.

- *Step 1.* Extract the relevant program snippet. The relevant program snippet of the given alarm statement is the backward program slice [18], which is constructed starting from the alarm statement and all the way back to the statement where the array or pointer access under the examination is safely allowed. To limit the area of code snippet, we restrict our backward slice only to grow back to the beginning of the current procedure. If buffer accesses are not safe even after we extended its backward slice to the beginning of the procedure, we abort the examination for the alarm.
- *Step 2.* Build initial context formulae. During the backward slicing for the extraction of the relevant program snippet, the slice could stop growing at the statement where all the initializations of the needed variables are not reached yet. Since we use two stop conditions to backward slicing, the slice can stop growing when it finds the buffer access under examination is safe or it encounters the beginning of the current procedure. As a result, live-in variables to the program snippet require their initial values in order to perform symbolic execution. The initial context for such live-in variables can be constructed from the interval results from the first phase, our buffer overflow analysis. For example, a variable `i` at the beginning of the relevant program snippet has an interval [0, 10] as a result of the first phase analysis. This interval can be converted to a formula, $(0 < \texttt{i}) \wedge (\texttt{i} < 10)$, since the interval actually represents that the minimum value of `i` is 0 and the maximum value of `i` is 10.
- *Step 3.* Translate the relevant program snippet into SMT formulae. We convert the relevant program snippet into a single static assignment form, then we translate the SSA form into SMT formulae. Thanks to the expressiveness of SMT formulae, translation rules are straightforward in most cases. If we encounter a statement which cannot be translated to SMT formulae, we mimic the statement by using the interval results of the first phase analysis. For example, we currently do not translate bitwise operations into SMT formulae. However we have implemented abstract versions of such bitwise operations in Raccoon, Raccoon's analysis results often suggest more precise constraints after bitwise operations. A procedure call is another example that need to be mimicked instead of performing the symbolic execution for the procedure body. In other words, we supply the SMT formulae with the resulting intervals of the procedure call, which is taken from the first phase analysis. The complete translation rules are presented in Section 4.
- *Step 4.* Symbolic execution of the relevant program snippet over an SMT solver. If the relevant program snippet is just a sequence of simple expressions, all we need to do is to verify the satisfiability for the conjunction of the initial context formulae, translated formulae for the relevant program snippet, and the error constraint at the alarm statement. For loop statements, however, we need a systematic way to construct all feasible execution paths. Our filtering phase uses a continuous loop *unrolling* technique to handle loop statements. The key idea of our unrolling is to linearize each execution path context one by one. The unrolling in our paper does not mean the loop transformation, which is used to increase the instruction level parallelism, but the resulting body of the loop has a similar form of our. The difference in ours is that we have to continuously unroll one iteration by one iteration until we reach to the upper bound of loop iteration condition and evaluate the all the possible execution paths of a loop. Fig. 2 shows an example of linearized contexts. In Fig. 2, α, l_i, and r_i represent the initial context, the leaving path context, and the remaining path context, respectively. The remaining path context represents the loop remaining condition plus the execution of loop body. The leaving path context represents the loop exit condition plus the statements outside the loop up to the statement right before the alarm statement. The i, where $i \in N$ and N is the number of iterations for the loop, represents the i_{th} iteration. Thus, the formula, l_i, represents the constraint that can be satisfiable when the loop is exited at the condition check before i_{th} iteration. The formula, r_i, represents the constraint when the i_{th} iteration is executed. The formulae, l_i^e and r_i^e, in Fig. 3 are the error conditions when the alarm statement under investigation is executed along the execution paths of l_i and r_i, respectively.
- *Step 5.* Filtering a false alarm
  - *Step 5a.* Loop handling when the alarm is outside the loop. As shown in the algorithm on the left side of Fig. 3, we can examine the given alarm statement along each piece of the linearized context, which is virtually the same as executing all the possible iterations of the loop. For an example, if the conjunction formula for the leaving path, $\alpha \wedge l_1 \wedge l_1^e$, is satisfiable, this means the error condition, l_1^e, can be realized along the context $\alpha \wedge l_1$. Thus, the alarm cannot be a false one. If there is a possibility of true alarm, the filtering analysis for this alarm stops without pursuing further investigation. Otherwise, the remaining path constraint, r_1 is added to the iteration context, $\alpha \wedge r_1$. If the added context is satisfiable, we recursively examine the next loop iteration. If the loop cannot be iterated further at the current iteration (i.e., $ctxt \wedge r_i$ is unsatisfiable), we have reached the end of loop iteration after executing all the possible number of iterations without finding any feasible error condition. Thus, we decide the alarm is false.
  - *Step 5b.* Loop handling when the alarm is inside the loop. As shown in the algorithm on the right side of Fig. 3, we examine the constraint including the error condition, $\alpha \wedge r_1 \wedge r_1^e$. If we find this formula is satisfiable, an error can be realized at the first iteration. Thus, this alarm is not a false alarm. We stop exploring the next iterations. Otherwise, we recursively examine the next iteration.

Note that we cannot conclude that the given alarm is a real error, when the error constraint is satisfiable. Although we do not use any abstraction in SMT translation and symbolic execution, the initial context formulae from the first phase analysis, buffer overflow analysis, may be over-approximated ones, which already contain infeasible states from the beginning. Thus, the symbolic execution of our filtering analysis still could find a satisfiable execution path for a false alarm.

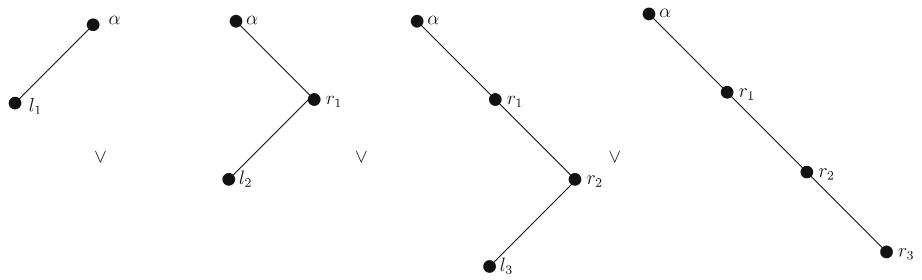

Fig. 2. An example of linearized contexts: the alarm to test is located outside the loop.

```
isFalseAlarm_outLoop($ctxt, i$)
(when alarm exists outside the loop)
1: if  sat($ctxt \wedge l_i \wedge l_i^e$)  then
2:     return  DONT_KNOW
3: else if  sat($ctxt \wedge r_i$)  then
4:     $nctxt = ctxt \wedge r_i$
5:     isFalseAlarm_outLoop($nctxt, i + 1$)
6: else
7:     return  YES
8: end if
```

```
isFalseAlarm_inLoop($ctxt, i$)
(when alarm exists inside the loop)
1: if  sat($ctxt \wedge r_i \wedge r_i^e$)  then
2:     return  DONT_KNOW
3: else if  sat($ctxt \wedge r_i$)  then
4:     $nctxt = ctxt \wedge r_i$
5:     isFalseAlarm_inLoop($nctxt, i + 1$)
6: else
7:     return  YES
8: end if
```

Fig. 3. Loop handling algorithm: Functions `isFalseAlarm_outLoop()` and `isFalseAlarm_inLoop()` return YES if all the alarms in a given program snippet are false positives and return *DONT_KNOW* if there exists a potentially real error. Depending on the location of the alarm to test, the appropriate function is used. Function `sat()` decides the satisfiability of the formula given as its parameter.

```
302:    count = lseek(tape, count, whence);
303:    if (count < 0)
304:       goto ioerror;
305:
306:    p = count_string + sizeof(count_string);
307:    *--p = '\0';
308:    do
309:       *--p = '\0' + (int)(count % 10);
310:    while ((count /= 10) != 0);
```

Fig. 4. A code snippet from rmt.c in GNU tar 1.13.

3. A concrete example

In this section, we provide the detailed description of our false alarm filtering with the example shown in Fig. 4. The example code is excerpted from rmt.c in GNU tar 1.13. When we ran our buffer overflow analyzer on this file, rmt.c in GNU tar 1.13, our analyzer produces a buffer underflow alarm at line 309. The reason for the reported alarm is that the size of the buffer is 64, while the possible values of offset lie in the interval $[-\infty, 62]$.

- *Step 1.* Extracting a relevant program snippet. At the first step of our filtering analysis, we need to select a program snippet which is relevant to the alarm statement to test. When we feed the file rmt.c of GNU tar program, our buffer overflow analysis finds multiple potential buffer overflow alarms. The example code in Fig. 4 has two memory references through pointers. The statements at line 307 and line 309 are the main targets for our buffer overflow analyzer to test. After performing points-to analysis, value analysis and buffer analysis in order, Raccoon, our buffer overflow analyzer, finally decides that the statement at line 307 is a safe access within buffer. Whereas, our analyzer decides that the statement at line 309 is potentially an out of range access and issues an alarm for the statement. Starting from the statement at line 309, our filtering analysis tracks the relevant program snippet. By using program slicing with slicing criterion (309, `p`), our analysis decides that the statements from line 308 to line 310 are the relevant program snippet to the alarm statement.
- *Step 2.* Building an initial context formulae. To perform symbolic execution for the extracted program snippet, we need value information for the live-in variables. The information about those scalar and pointer variables are available from the first phase analysis, buffer overflow analysis. Available information is stipulated as quantifier-free first-order logic. In our example, the information about variable `p` and `count` is stipulated as follows:

– The initial context α:

$$(\texttt{p.offset}_0 = 63) \wedge (\texttt{p.size}_0 = 64) \wedge (0 \leqslant \texttt{count}_0 \leqslant \texttt{LONG\_MAX})$$

To distinguish each occurrence of the same variable in SMT formulae, we rename the variable by adding different subscripts to the variable. Our renaming scheme is actually similar to that of static single assignment (SSA) transformation. The information for a scalar variable is given as an interval, minimum and maximum values. Meanwhile, the information for a pointer variable is a pair of two intervals: one for offset and the other for size. In the above SMT formulae, we specify the offset and the size of a pointer variable, `p` with an integer value each, but the actual information from the buffer overflow analysis is given as an interval for each. The value of `p.size` represents the allocated bytes of the buffer referenced by the pointer variable `p`. The pointer variable `p` could point to any position between the beginning and the end of the data. The value of `p.offset` represents the exact position of the data to which the pointer variable `p` points. Considering that the value of `p.offset` can start from zero, we represent the fact that the pointer variable `p` points to the last element of a buffer the size of which is 64 bytes long.

- *Step 3, 4.* Translation into SMT formulae and symbolic execution. The loop located at line 308 through line 310 is unrolled enough number of times so that all the execution paths for the loop can be considered. To accomplish the loop unrolling, the remaining path conditions are asserted along with other assertions imposed by the loop body statements. For example, line 310 of Fig. 4 imposes an assertion $(\texttt{count}_1 \neq 0)$ for the remaining path at the first iteration of the loop; other assertions for leaving path contexts and remaining path contexts at various iterations are as follows:

– leaving path context l_i at the ith iteration:

$$(\texttt{count}_i = 0)$$

– remaining path context r_i at the first iteration $(i = 1)$:

$$\wedge (\texttt{p.offset}_1 = \texttt{p.offset}_0 - 1) \wedge (\texttt{count}_1 = \texttt{count}_0/10)$$
$$\wedge (d(\texttt{p.base}_0 + \texttt{p.offset}_1) = 48 + \texttt{count}_0\%10)$$

– remaining path context r_i at the ith iteration:

$$(\texttt{count}\ _{i-1}{\neq}0)$$
$$\wedge (\texttt{p.offset}_i = \texttt{p.offset}_{i-1} - 1) \wedge (\texttt{count}_i = \texttt{count}_{i-1}/10)$$
$$\wedge (d(\texttt{p.base}_0 + \texttt{p.offset}_i) = 48 + \texttt{count}_{i-1}\%10)$$

Pointer dereferences are translated with the uninterpreted function d which is an element of our SMT formulae. After the loop is unrolled enough number of times, the remaining path condition cannot be satisfied anymore. Intuitively, the loop execution procedure unrolls the loop until such inconsistency is found by the SMT solver.

- *Step 5*. Filtering a false alarm. By investigating the reachability to error states from initial states, we check if a given alarm is genuine or not. If error states are not reachable, the given alarm is a false positive. In our example, the alarm statement at line 309 is on one of the remaining paths, an error constraint formula l_i^e on an execution path l_i should be *false*. An error state on the remaining paths should satisfy the following assertions:
  – error constraint r_1^e on the execution path r_1:

$$((\texttt{p.offset}\ _1 \geqslant \texttt{p.size}\ _0) \vee (\texttt{p.offset}\ _1 < 0)) \wedge (\texttt{p.offset}\ _1 \leqslant 62)$$

  – error constraint r_i^e on the execution path r_i:

$$((\texttt{p.offset}\ _n \geqslant \texttt{p.size}\ _0) \vee (\texttt{p.offset}\ _n < 0)) \wedge (\texttt{p.offset}\ _n \leqslant 62)$$

The first conjunct asserts such condition that the pointer access by `p` is safe. The second conjunct confines a set of the error states by taking advantage of the result from the previous buffer overflow analysis, which dictates the interval of `p.offset` is $[-\infty, 62]$.

Finally, with the formulae listed in the above, α, l_i, r_i, l_i^e and r_i^e, the procedure `isFalseAlarm_inLoop`$(\alpha, 0)$ in Fig. 3 returns `YES` after unrolling the loop 18 times, and thus the given alarm can be filtered out.

4. Transforming C program snippets into SMT formulae

In this section, we explain how we can translate C program snippets into SMT formulae. We used the CIL infrastructure [14] to implement our analyzers. The CIL features a reduced number of syntactic and conceptual forms. Several ANSI-C syntactic statements are translated into equivalent ones during the CIL transformation. For example, all looping constructs (i.e., *for, while, do-while*) are converted to a single form. The CIL expressions have no side effects. In other words, the side effects occur only with assignment statements.

4.1. Extraction of relevant program snippet

A program snippet relevant to an alarm is basically a backward program slice constructed from that alarm statement. In many cases, we do not need a complete program slice to prove the safety of a buffer access under investigation. We force a relevant program snippet to begin with the point where the target buffer can be accessed safely. In other words, we track back to the point where we can prove that the target buffer access is safe only with the information from the first phase analysis. Moreover, we prohibit the extraction process from tracking back beyond the procedure boundary. If a target buffer access is still unsafe at the entry of the current procedure (that directly contains the target alarm), we decide not to apply our filtering analysis for that alarm statement. If we allow to track the relevant statements beyond the program boundaries, we may end up with multiple program snippets for an alarm statement. Since the current procedure may be called from multiple call sites, the back-tracking process would produce multiple program snippets along the different call sites. In such situation, our alarm filtering phase may need to investigate too many program snippets. To avoid such implications and their overhead, we limit the range of the relevant statement tracking within the current procedure.

4.2. Alarm grouping

When multiple alarms share the same program snippet, we can reduce the analysis time of the filtering phase by analyzing those alarms all together simultaneously. Our alarm grouping is performed with the following two heuristics. (1) If a relevant program snippet for an alarm A contains another buffer access, which corresponds to another alarm B, then the alarm A and the alarm B are grouped together. (2) Two alarm groups have common alarms, all the alarms in both alarm groups are grouped together. To handle a group of alarms, the functions `isFalseAlarm_*` in Fig. 3 is modified so that the functions decide the satisfiability for each alarm in the group. Instead of stopping the procedure and returning *DONT_KNOW* for one alarm, the modified function checks out the satisfiability of the rest of alarms. In the experimental section, we measure the times spent in the filtering phase by using alarm grouping scheme.

4.3. Translation rules

A subset of a C-like language with guarded assignments, as shown in Table 1, is considered for the translation rules. A program variable `v` is renamed to $\texttt{v}_n$ where n denotes the number of assignments made to the variable `v` prior to the given program point. Although such renaming technique is similar to the static single assignment (SSA) transformation, our transformation is different in that we do not need ϕ-functions during our transformation. All the expressions we use have no side effects. The side effects only occur at assignment statements. An assignment statement executes only when the boolean expression in the guard can be

Table 1
Source language.

$$\begin{array}{rl}
\langle \mathrm{exp} \rangle ::= & \langle \mathrm{const} \rangle \\
& | \langle \mathrm{lval} \rangle \\
& | \& \langle \mathrm{lval} \rangle \\
& | - \langle \mathrm{exp} \rangle \\
& | \langle \mathrm{exp} \rangle_1 \, op_b \, \langle \mathrm{exp} \rangle_2 \\
\langle \mathrm{lval} \rangle ::= & \langle \mathrm{var} \rangle | * \langle \mathrm{var} \rangle \\
\langle \mathrm{stm} \rangle ::= & \langle \mathrm{stm} \rangle_1 ; \langle \mathrm{stm} \rangle_2 \\
& | \langle \mathrm{bexp} \rangle \rightarrow \langle \mathrm{lval} \rangle := \langle \mathrm{exp} \rangle \\
& | \texttt{assume} \langle \mathrm{bexp} \rangle
\end{array}$$

Table 2
Translation rules for expressions and statements.

Source	SMT formula
(a) Translation of *L*-values	
$T_{lval}(v)$ where $v \in \langle\text{var}\rangle$	$N(v)$
$T_{lval}(*v)$	$d(T_{lval.base}(v) + T_{lval.offset}(v))$
$T_{lval.base}(v)$ where $v \in \langle\text{pointer}\rangle$	$N(v)$.base
$T_{lval.offset}(v)$	$N(v)$.offset
$T_{lval.length}(v)$	$N(v)$.length
$T_{lval.size}(v)$	$N(v)$.size
(b) Translation of expressions	
$T_{exp}(c)$ where $c \in \langle\text{const}\rangle$	c
$T_{exp}(lv)$ where $lv \in \langle\text{lval}\rangle$	$T_{lval}(lv)$
$T_{exp}(-e)$	$-T_{exp}(e)$
$T_{exp}(e_1 op_b e_2)$ where $op_b \in \{+,-,*\}$	$T_{exp}(e_1) op_b T_{exp}(e_2)$
$T_{exp}(e_1 \% e_2)$	r
$T_{exp}(e_1 / e_2)$	q
Auxiliary formulae for % and /	$d = T_{exp}(e_2) \wedge (T_{exp}(e_1) = d * q + r) \wedge$ $(0 \leqslant r < d)$ where d, r, and q are new
$T_{exp.base}(str)$ where $str \in \langle\text{string}\rangle$	$base(str)$
$T_{exp.offset}(str)$	0
$T_{exp.length}(str)$	$length(str)$
$T_{exp.size}(str)$	$length(str) + 1$
$T_{exp.base}(arr)$ where $arr \in \langle\text{array}\rangle$	$N(arr)$.base
$T_{exp.offset}(arr)$	0
$T_{exp.length}(arr)$	$N(arr)$.length
$T_{exp.size}(arr)$	$sizeof(arr)$
$T_{exp.base}(lv + e)$ where $lv \in \langle\text{pointer}\rangle$	$T_{lval.base}(lv)$
$T_{exp.offset}(lv + e)$	$T_{lval.offset}(lv) + T_{exp}(e) *\ unitsize(lv)$
$T_{exp.length}(lv + e)$	$T_{lval.length}(lv)$
$T_{exp.size}(lv + e)$	$T_{lval.size}(lv)$
(c) Translation of statements	
$T_{stm}(s_1; s_2)$	$T_{stm}(s_1) \wedge T_{stm}(s_2)$
$T_{stm}(g \rightarrow lv_1 := \&lv_2)$	$g \Rightarrow (T_{exp}(*lv_1) = T_{exp}(lv_2))$
$T_{stm}(g \rightarrow lv := e)$	$g \Rightarrow (T_{lval}(lv) = T_{exp}(e))$
$T_{stm}(g \rightarrow lv := e)$ where $lv \in \langle\text{pointer}\rangle$	$g \Rightarrow (T_{lval.base}(lv) = T_{exp.base}(e)$ $\wedge T_{lval.offset}(lv) = T_{exp.offset}(e)$ $\wedge T_{lval.length}(lv) = T_{exp.length}(e)$ $\wedge T_{lval.size}(lv) = T_{exp.size}(e))$
$T_{stm}(\texttt{assume}\ b)$	b

satisfied. On the other hand, we use a quantifier-free first-order logic as our target language.

Table 2 exhibits how we translate a source program snippet into its corresponding target SMT formula. First, a sequence of statements is translated into a conjunction of translations of sub-statements. Then, we classify assignments into two types: assignments with & operators and simple assignments. An assignment with & operator generates formulae for the dereference of the l-value, while a simple assignment produces formulae for the l-value itself. Guarded assignments can be easily translated by using implications. In this paper, we do not consider interprocedural analysis for the filtering phase. Although we ignore function calls in the filtering phase, we can make use of the information from the prior buffer overflow analysis phase. That means we can have the upper and lower bounds of a procedure, when its return value is of scalar type. We also assume statements, which have live-in variables inside, encode the information about the live-in variables from the prior buffer overflow analysis and generate approximated formulae for the SMT solver.

Simple variable accesses are translated by using the naming function *N*. The naming function *N* returns a unique identifier if the variable name passed onto the function is never used before. Otherwise, the identifier associated with the variable name is returned. Pointer dereferences are translated with the uninterpreted function *d*. The uninterpreted function *d* represents a dereference of its associated l-value, and the uninterpreted function *d* itself is an element of our SMT formulae. For example, a source program, $p = \&x;\ y = *p$, is translated into $d(p) = id_x \wedge id_y = d(p)$ where id_x and id_y are unique identifiers for variable *x* and *y*. This translated formula can be used to infer $id_x = id_y$.

Unary operators and binary operators are translated into corresponding operators in Yices, the SMT solver we use. Although Yices does not directly support integer division and remainder operations, we are able to translate such expressions by using integer multiplication; for example, $x = y/z$ can be translated into $(x = q) \wedge (y = z * q + r) \wedge (0 \leqslant r < z)$ where *r* and *q* are new integer variables.

4.4. Translation of function calls

In the false alarm filtering phase, we take care of function calls within program snippets with a different scheme from simple statements. One option would be to employ powerful interprocedural analysis and apply symbolic execution to the body of the function. This scheme would incur too much overhead for our filtering analysis. Thus, our filtering analysis does not resort to a complex interprocedural analysis. Instead, we retrieve the summarized information of the procedure from Raccoon, our first phase buffer overflow analysis. When we encounter a function call during the program slicing, we produce assumptions on the affected global variables in forms of SMT formulae. We also produce an assumption on the return value of the function by importing the analysis results from the first phase buffer overflow analysis.

When we transform an unrolled program snippet to a more manageable form for the SMT translation, we perform the transformation of the call sites first. The transformation needed for a call site is to insert assignments to global variables. Assume, for an example, a function `foo()` and its subsequent functions called inside use a global variable `g` the type of which is integer. Then, we place an assignment to that global variable `g` after the call site `foo()` to reflect the change made from the function call.

```
assume (from_foo@l_g >= min and from_foo@l_g <= max);
g := from_foo@l_g
```

In the above formula, `from_foo@l_g` is a unique temporary variable that is supposed to hold the value of `g` when `foo()` returns. Two values, *min* and *max*, represent the minimal and maximal value of `from_foo@l_g`, respectively. These values are available from the first phase of buffer overflow analysis. If a procedure call has a parameter of pointer type, the dereference of this parameter also has to be updated after the call site. For instance, a call site `foo(p)` has a parameter `p` and the type of `p` is a pointer to an integer value. Then, we need to add an update assignment to `p` after the call site, `foo()`.

If the return value of a call is assigned to a variable, an additional transformation is applied. For example, if a call site assigns its return value to a variable `v`, as in `v:=foo()`, the assignment is transformed to:

```
assume(foo@l_ret >= min and foo@l_ret <= max);
v := foo@l_ret
```

In the above formula, `foo@l_ret` is a unique temporary variable that is expected to hold the return value of `foo()`. The two values, *min* and *max*, represent the minimal and maximal return value, respectively. These values are also available from the first phase of our buffer overflow analysis.

5. Experimental results

In our experiment, we investigate all the alarms of the target programs in order to measure the effectiveness and limitation of our filtering scheme. We first analyze the target programs with Raccoon, our first phase analyzer. In the second phase, for each buffer overflow alarm, the relevant program snippet containing the alarm is extracted, and the alarms sharing the same relevant program snippet are grouped together. For each program snippet, it is automatically translated into SMT formulae according to the translation rules described in Section 4.

We use Raccoon [11] as the base buffer overflow analyzer and Yices 1.0.16 [2] as the SMT solver for symbolic execution. Thanks to SWIG [1], we are able to connect Yices C API with Raccoon's OCaml code without too much difficulty. Yices API includes dynamic features such as storing current state, dynamically adding new assertions, and rollbacking to a certain point. With such functionality, we are able to efficiently implement our loop execution algorithm shown in Fig. 3. Our experiments are performed on a PC with two 2.33 GHz quad-core XEON processors and 8 GB of memory.

5.1. Target benchmarks

For evaluation purpose, we use a set of programs introduced in [20]. The set of programs reproduces the buffer overflow vulnerabilities which are found in the earlier versions of three open source applications: *bind*, *sendmail*, and *wu-ftpd*. Instead of whole source programs, the reproduced programs contain only the related functions within a file, but if multiple types of vulnerabilities exist, separate files contain source code for each type of vulnerability. Table 3 shows the four target programs from bind, seven target programs from sendmail, and three target programs from wu-ftpd. It also describes the source of buffer overflow vulnerabilities in each target program.

Applying interprocedural analyses, our first phase analysis can handle large C programs within tens of minutes [11]. Meanwhile, the second phase applies an intra-procedural analysis to filter out the alarms from the first phase. That means our filtering scheme works within procedure boundaries. Thus, applying our filtering scheme to the target programs, which reproduce the original bugs of the real-world applications in small sized programs, is a valid approach to evaluate our second phase, the intra-procedural filtering scheme. When we extract relevant program snippets, we limit ourselves to the boundaries of procedures. Thus, the size of the whole source code rarely matters. Even if the target programs are small, they are enough to show the capability of our alarm filtering scheme.

The vulnerability reproduced in BIND-1 and BIND-3 is related to `memcpy`. If this function is invoked with a *size* argument larger than the size of the target buffer, a buffer overflow occurs. In BIND-2, a negative *size* argument for `memcpy` can underflow to a large positive value, which again causes a buffer overflow in the target buffer. In BIND-4, the `sprintf` function can make the target string longer and overflow the target buffer which is provided to store the generated string.

SM-1 changes the position of pointer, which points to the upper bound of the target buffer, according to the input character stream. Correct code should decrease the upper bound on '<' input and increase the upper bound on '>' input, but SM-1 does not perform decreasing operations for '<' input. As a result, the pointer to the upper bound mistakenly points to a wrong place. In SM-2, `strcpy` read the source string from a file but its size can be longer than the size of the target buffer. In SM-3, a pointer to the output buffer should be initialized at the end of line so that it points to the beginning of the target buffer again, but the code fails to do so. Thus, the pointer only increases without reset and makes the output buffer overflow after reading more than MAXLINE bytes from a file. In SM-4, a necessary size check is not performed due to a typo, which instead causes a fault in the size check expression. In SM-5, when an input byte, 0xff, is copied into a *signed char* variable, it is evaluated as -1 meaning an internal error code (NOCHAR); a multiple inputs of 0xff can cause buffer overflows. In SM-6, when an unsigned integer value is copied into the signed index variable, it can be evaluated as a negative value via integer overflows; negative values silently pass the size check and cause buffer underflows. In SM-7, `strncpy` read the *size* argument from a packet header and the read value can be larger than the size of the target buffer.

In FTP-1 and FTP-3, `strcpy` and `strcat` is called without checking the size of source string. As a result, these calls can overflow the target buffers if the size of source is larger than that of target. In FTP-2, a mistakenly coded size check can cause a buffer overflow in `strcat`. This size check should have used '>=' instead of '>' in its comparison expression.

Table 4 classifies the alarm groups into five categories. The first column shows the names of target programs, and the second column shows the names of categories. The third and the fourth columns show the total number of alarm groups and the total number alarms in each category, respectively. If our filtering scheme proves the condition of the alarms are not feasible, we categorize them into *false positives*. If our scheme cannot prove the alarms are false, we categorize them into *possible overflows*. When we extract the relevant program snippet, we could reach the beginning of the procedure without finding the point where the buffer access is safe. In this case, we need not to proceed further and categorize these alarms into *interprocedural* since we may need interprocedural filtering analysis for them. For some program snippets, our filtering scheme takes too long or consumes too much space due to their complex iteration structures and control flows. In this case, our filtering scheme bails out without finishing the symbolic execution. We categorize these alarms into *time-out*. In addition, we do not consider the buffer overflow alarms occurred in C library calls such as `strcpy` and `memcpy`. Currently, our analysis cannot handle the alarms originated from the library functions with two buffers, yet. For these alarms, we categorize them together into *library calls*. For these alarms, we need not to extract and group program snippets. Thus, the number of groups in Table 4 is not specified at all. Finally, since our analysis does not model integer overflows/underflows, our filtering scheme cannot handle the cases where integer overflows are the causes of buffer overflows. We categorize such alarms into *integer overflows*. Among all the categories, only the alarms in the *false positives* category can be filtered out with our filtering analysis.

Table 5 summarizes the experimental results with a set of target programs which are reproduced from three open source

Table 3
Known vulnerabilities in target programs.

Target	Reason
BIND-1	A size argument of `memcpy` is not checked
BIND-2	A negative argument to `memcpy` underflows to large positive int
BIND-3	A size argument of `memcpy` is not checked
BIND-4	Unchecked `sprintf` calls
SM-1	The upper bound is incremented for '>' but not decremented for '<'
SM-2	`gecos` field copied into fixed-size buffer without size check
SM-3	A pointer to a buffer is not reset to the beginning at the end of a line
SM-4	A typo prevents a size check from being performed
SM-5	An input byte, 0xff, is erroneously cast to -1, an error code
SM-6	Negative indexes passe size check but cause underflow
SM-7	A size argument of `strncpy` is read from a packet header but not checked
FTP-1	Several `strcpy` calls without bounds checks
FTP-2	A wrong size check inside `if`. > should really be >=
FTP-3	Unchecked `strcpy` and `strcat` calls

Table 4
Categorization of buffer-overflow alarm groups.

Target	Category	# Groups	# Alarms
BIND-1	False positives	1	29
	Library calls	–	1
BIND-2	False positives	1	35
	Library calls	–	1
BIND-3	False positives	1	1
	Library calls	–	1
BIND-4	False positives	1	1
	Library calls	–	1
SM-1	Time out	1	28
SM-2	Possible overflows	2	2
	Interprocedural	4	4
	Library calls	–	5
SM-3	Interprocedural	3	3
SM-4	Time out	1	7
SM-5	Interprocedural	3	2
	Integer overflows	4	4
	Library calls	–	1
SM-6	Integer overflows	1	1
SM-7	False positives	2	20
	Possible overflows	1	2
	Interprocedural	2	17
	Library calls	–	3
FTP-1	False positives	1	1
	Library calls	–	6
FTP-2	False positives	1	1
	Library calls	–	5
FTP-3	False positives	1	1
	Library calls	–	23

programs with intentional code vulnerabilities. The first column shows the names of programs. The second column shows the number of source lines of code after we translated the programs into CIL code. To present various code vulnerabilities, parts of three well-known open source programs are implemented under different program names. The lines of code range about 200–1000 lines. The columns labeled *# Alarms* and *# False Alarms* show the total number of alarms reported by our buffer overflow analyzer and the total number of false positives manually identified by close inspection, respectively. The columns labeled *# Filtering Groups* shows the number of the relevant program snippets after grouping program snippets. The column labeled *# Filtered Alarms* shows the number of alarms that can be filtered out by our filtering analysis. The last two columns show the analysis times of Racoon, the first phase buffer overflow analysis, and Yices SMT solver, the second phase alarm filtering analysis. The detailed discussion is presented for each program in the following sections.

5.2. Evaluation of bind

In the target programs reproduced from *bind*, we are able to filter out all of 66 false alarms. The false alarms in the *bind* programs are originated from our fast buffer overflow analysis. In BIND-1 case, the first buffer overflow alarm occurs at a loop which involves two pointer variables. Static analyses based on abstract interpretation often employ widening and narrowing techniques to analyze loops efficiently and guarantee their termination. During the first phase analysis, both pointer variables are widened, but only the one pointer variable involved in the loop condition is narrowed. Although relational analyses such as [13] can recover the information for the other pointer variable from the relationship to the pointer variable in the loop condition, we cannot afford to accommodate such complex analyses within our first phase analysis. Since our first phase analysis is designed to verify the majority of simple parts of code as fast as possible, we rather employ a fast and light-weighted buffer overflow analysis. The inaccuracy of our fast analysis is actually the origins of the rest 28 alarms in BIND-1. Analyzing the relevant program snippet by our filtering algorithm, we are able to prove all buffer overflow alarms are false positives. The origins of the false alarms in BIND-2, BIND-3, and BIND-4 cases are similar.

The filtering times for BIND-1 and BIND-2 cases are much longer than other cases. The relevant program snippets for BIND-1 and BIND-2 cases span through a sequence of multiple loops. Since our filtering scheme unrolls loops and explores all the possible execution paths, multiple loops usually incur a long analysis time for the symbolic execution in SMT solver. On the other hand, the relevant program snippets for BIND-3 and BIND-4 cases contain only one loop. The case in BIND-3 unrolls its loop 10,000 times until the filter analysis proves it is a false positive. Meanwhile, the case in BIND-4 unrolls its loop only once to prove.

The analysis times for BIND-1, BIND-2, and BIND-3 is relatively high. It suggests that the current implementation of our filtering analysis is not optimized sufficiently. One of possible reasons is that the number of SMT variables in the automatic translation tends to be large. Excessive variable usages surely deteriorate the performance of SMT solver, particularly when such fragment of SMT formulae reused thousands of times within loops.

5.3. Evaluation of sendmail

In the target programs reproduced from *sendmail*, our filtering scheme is unsuccessful in filtering out false alarms except SM-7. In SM-1 case, 28 alarms are grouped together into one filtering group, resulting in the relevant program snippet which consists of 428 lines of CIL code. The current implementation of our filtering algorithm fails to finish the proof within reasonable time bound due to the complex control flows of the program snippet. In SM-2 case, our filtering scheme fails to prove any alarms, the seven of which are actually false. For four filtering groups, we fails to find that buffer accesses are safe even at the beginning of the procedure. Thus, our scheme skips the four filtering groups. The relevant program snippet for one group contains nested loops. Our filtering algorithm does not unroll an inner loop but adopts the analysis results from Raccoon. Still, at the end of the symbolic execution, our analysis cannot prove this alarm is false. In SM-3 case, the condition for the output buffer access is still unsafe even at the beginning of the procedure due to the imprecise analysis result from Raccoon. Thus, our filtering algorithm skips those three alarms. As for SM-4 case, we encounter an infinite loop within our symbolic execution since the termination condition of that loop is determined by a user input. Since we set the time bound for the SMT solver, our symbolic execution bails out after a predefined threshold time. When a loop termination is decided by input values, our filtering scheme assumes all the range of possible values according to the variable type. Thus, it incurs too many possible execution paths to handle within the time bound.

In SM-5 and SM-6 case, real buffer underflows occur by integer overflows. In SM-5 case, the special input character, 0xff, is copied into a signed variable and the value is evaluated to -1 which is an internal control code (NOCHAR). Such unexpected conversion breaks the input algorithm and it is possible to overflow the input buffer with multiple inputs of 0xff. In SM-6 case, a large positive number of unsigned type is copied into a signed index variable. This may also result in a negative number by integer overflows. If a negative number is used as an array index, it would cause a real buffer underflow. Our filtering analysis is not capable of handling integer overflows yet. Even though our buffer overflow analyzer detects the exact overflow points, our filtering analysis cannot handle them.

In SM-7 case, we are able to filter out 20 false alarms from two filtering groups. The removed false alarms are similar to those in

Table 5
Effect of alarm filtering: our filtering method removes 68% of false alarms on average.

Target	# CIL Lines	# Alarms	# False alarms	# Filtering groups	# Filtered alarms	Raccoon time (s)	Filtering time
BIND-1	806	30	29	1	29	0.05	584.95 s
BIND-2	999	36	35	1	35	0.10	1816.44 s
BIND-3	225	2	1	1	1	0.01	90.78 s
BIND-4	356	3	1	1	1	0.02	0.01 s
SM-1	541	28	0	1	0	0.14	Time out
SM-2	354	11	7	6	0	0.02	0.10 s
SM-3	392	3	0	3	0	0.01	0.03 s
SM-4	426	7	0	1	0	0.01	Time out
SM-5	480	7	4	6	0	0.04	N/A
SM-6	188	1	0	1	0	0.01	N/A
SM-7	752	42	40	4	20	0.05	3.03s
FTP-1	220	7	3	1	1	0.01	0.01 s
FTP-2	625	6	4	1	1	0.12	0.21 s
FTP-3	402	24	6	1	1	0.02	0.07 s
Total	6766	207	130	29	89	0.61	

BIND-1. Two related variables are involved in a loop and only one is narrowed in our buffer overflow analysis, but the symbolic execution based on SMT solver proves those alarms are false positives. Some false alarms still cannot be removed with our filtering scheme. They need an interprocedural scheme to filter out, but our filtering scheme works within procedure boundaries. For any live-in variables from outside the procedure, it relies on the analysis results from the first phase. Due to the imprecise analysis results of Raccoon around the function boundaries, our symbolic execution cannot filer out some of those false alarms.

5.4. Evaluation of Wu-ftpd

In FTP-1 case, a pointer is conditionally increased by one byte. If we employed a path sensitive analysis, we would have excluded such false alarm in the first phase. The analysis time required in such analysis, however, would be much longer than our SMT solver based filtering scheme. On the other hand, our symbolic execution with SMT solver filters out the false alarm within a second. Similarly to FTP-1, a pointer is conditionally decreased in FTP-2 case. Without a path sensitive analysis, our first phase reports an alarm for this access, but this is actually a false positive. Our filtering algorithm removed this false alarm by handling the loop precisely. In FTP-3 case, the alarm detected by the first phase is a similar case to BIND-1, where two pointer variables are involved within a loop. This alarm is filtered out by the second phase symbolic execution as in BIND-1 case.

6. Related work

Unlike our false-alarm filtering method relying on symbolic execution, false-alarm filtering based on abstract interpretation is also possible as reported by Rival [15]. Rival's method essentially calculates the intersection of two kinds of abstract data, one computed by usual forward abstract interpretation and the other one computed by performing *backward* abstract interpretation starting from the alarm program point with the initial condition to be the alarm condition. If this intersection is empty at an alarm program point, it can be said that the alarm is false. Although it is interesting to see how abstract interpretation can remedy its weakness such as false alarms on its own, backward abstract interpretation shares the common weakness of abstract interpretation, i.e., precision loss. Therefore, the false-alarm filtering with capability of backward abstract interpretation may not exceed the capability of symbolic execution.

Indeed, to compensate for this precision loss, Rival proposed to constrain an *execution pattern* and *input* when backward abstract interpretation fails to determine the falsity of an alarm. While Rival left the automation of this constraint as future work, Gulavani and Rajamani proposed a method that can automatically constrain an execution pattern into limited forms [9]. They focused on restricting an execution pattern of a loop so that widening can be performed after a few iterations of the loop body. It is well-known that widening is a major cause of precision loss, and precision is sometimes improved by deferring widening. Their method automatically detects the widening operation that causes a false alarm, and defers this widening. While their method can filter out some of false alarms reported due to too early widening, it was also pointed out by the authors that their method has the following limitations. First, widening is not monotonic. Therefore, precision at some cases deteriorates after deferring widening operation on the contrary to one's expectation. Second, widening deferment should be repeated until the falsity of an alarm can be determined, and this repetition is not guaranteed to terminate.

In comparison to the above abstract-interpretation-based methods, symbolic execution simply considers all the feasible executions and input. Despite its relatively high overhead, the empirical results of our own demonstrates that computer hardware and supportive tools such as SMT solvers are mature enough to apply symbolic execution to small program regions around alarm points to filter out false alarms.

As mentioned earlier, we do not aim to determine the truth of alarms. However, note that it is possible to identify true alarms if we apply symbolic execution to a program snippet between the beginning of a program and the alarm point. Indeed, the Erez's true-alarm detecting method [7] performs backward symbolic execution, that is, weakest precondition generation, and checks if the weakest precondition at the beginning of a program is satisfiable to determine the truth of the alarm. We opt for investigating small program regions through symbolic execution, because we believe that it is more helpful to filter out as much false alarms as possible at the soonest possible time than to assert that some alarms are true. Once an alarm list of reasonable length is ready, the truth of each alarm can be determined with the help of other methods specialized for it such as model checking.

On the other hand, there are some statistical approaches to show alarms most likely to be real errors over those that are least likely. Z-ranking [12] is a statistical technique to rank alarms from most to least probable based on frequency counts of successful and failed checks. The basic idea is that an always fail case is highly likely to be a false positive, so many successful checks and a few failures lead to high ranking. They applied z-ranking to lock errors, free errors, and string format errors. Airac [10] adopts a Baysian approach to rank alarms based on the predefined set of symptoms, which describe the analyzer's internal behaviors and the input programs' syntactic contexts. Each alarm is characterized by a set of symptoms and the user's judgment about an alarm automatically adjusts the probabilistic influence of each symptom. After a certain number of training judgments, the system is able to decide the probability of a new alarm based on symptoms. Airac's Baysian approach is designed for buffer overflow analysis, and we believe that z-ranking can be extended for buffer overflow analysis. Since these statistical approaches are orthogonal to ours, they can be integrated into our system for more useful results.

7. Conclusion

Static verification tools based on sound analysis techniques often suffer from the false alarm problem. Although the more precise analysis generally results in fewer false alarms, the cost to perform such precise analysis is often too high to be practical for large software. After investigating several open source programs, we notice that large parts of the code can be verified with relatively imprecise analysis and only small fragments of the code actually need expensive but precise analysis.

In this article, we present our two phase analysis technique to detect buffer overflow defects. Our first analyzer, Raccoon is able to finish three analyses within a few hundreds of seconds for ten thousand lines of code from several open source programs and to prove that about half of array and pointer accesses are safe. Taking advantages of a state-of-the-art SMT solver, our second phase performs more precise analysis on a program snippet which triggers a buffer overflow alarm in the first phase.

We have implemented our symbolic execution scheme for alarm filtering using Yices and show that our filtering method can effectively remove a considerable number of false alarms. Our experiment with the multiple cases, which are reproduced from three open source programs, shows that our filtering method can reduce 68% of false alarms on average. In addition to that, we investigate each case in detail and provide our insight on the cases where our filtering scheme successfully works and also the cases it fails to filter out.

Acknowledgement

This research was supported by the Korea Research Foundation Grant funded by the Korean Government (MEST, Basic Research Promotion Fund, KRF-2007-331-D00427).

References

[1] Simplified Wrapper and Interface Generation <http://www.swig.org/>.
[2] Yices: an SMT solver <http://yices.csl.sri.com/>.
[3] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, POPL, 1977. pp. 238–252.
[4] P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among variables of a program, POPL, 1978. pp. 84–96.
[5] M. Das, Unification-based pointer analysis with directional assignments, PLDI, 2000. pp. 35–46.
[6] N. Dor, M. Rodeh, S. Sagiv, Cssv: towards a realistic tool for statically detecting all buffer overflows in c, PLDI, 2003. pp. 155–167.
[7] G. Erez, Generating concrete counterexamples for sound abstract interpretation, Master's thesis, Tel-Aviv University, Israel, 2004.
[8] D. Evans, D. Larochelle, Improving security using extensible lightweight static analysis, IEEE Software 19 (1) (2002) 42–51.
[9] B.S. Gulavani, S.K. Rajamani, Counterexample driven refinement for abstract interpretation, in: Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06), LNCS, vol. 3920, Springer, 2006, pp. 474–488.
[10] Y. Jung, J. Kim, J. Shin, K. Yi, Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis, SAS, 2005. pp. 203–217.
[11] Y. Kim, J. Jeon, H. Han, Development of cost-effective buffer overrun analyzer, KIISE SIGPL Transactions on Programming Languages 19 (2) (2005) 1–9.
[12] T. Kremenek, D.R. Engler, Z-ranking: Using statistical analysis to counter the impact of static analysis approximations, SAS, 2003. pp. 295–315.
[13] A. Miné, A new numerical abstract domain based on difference-bound matrices, in: Proc. of the 2d Symp. on Programs as Data Objects (PADO II), Lecture Notes in Computer Science, vol. 2053, Springer, Aarhus, Danemark, 2001, pp. 155–172. <http://www.di.ens.fr/mine/publi/article-mine-padoII.pdf>.
[14] G.C. Necula, S. McPeak, S.P. Rahul, W. Weimer, Cil: intermediate language and tools for analysis and transformation of c programs, CC, 2002. pp. 213–228.
[15] X. Rival, Understanding the origin of alarms in astrée, in: Proceedings of the Second International Symposium on Static Analysis (SAS'05), LNCS, vol. 3672, Springer, 2005, pp. 303–319.
[16] A. Venet, G.P. Brat, Precise and efficient static array bound checking for large embedded c programs, PLDI, 2004. pp. 231–242.
[17] D. Wagner, J.S. Foster, E.A. Brewer, A. Aiken, A first step towards automated detection of buffer overrun vulnerabilities, NDSS, 2000. pp. 3–17.
[18] M. Weiser, Program slicing, ICSE, 1981. pp. 439–449.
[19] Y. Xie, A. Chou, D.R. Engler, Archer: using symbolic, path-sensitive analysis to detect memory access errors, ESEC/SIGSOFT FSE, 2003. pp. 327–336.
[20] M. Zitser, R. Lippmann, T. Leek, Testing static analysis tools using exploitable buffer overflows from open source code, in: R.N. Taylor, M.B. Dwyer (Eds.), SIGSOFT FSE, ACM, 2004, pp. 97–106.