

Buffer Overflow Detection using Environment Refinement

Franjo Ivančić, Sriram Sankaranarayanan, Aarti Gupta

NEC Labs America

`ivancic,srirams,agupta@nec-labs.com`

and

Ilya Shlyakhter

Broad Institute, MIT.

`ilya_shl@alum.mit.edu`

The static detection of buffer overflow bugs has received much attention, recently. Whereas, expensive techniques are not scalable, lightweight approaches present too many false positives. In our experience, approaches such as predicate abstraction and refinement that are useful for finding real API-usage bugs in low-level C programs have proved inadequate for this problem.

We present a technique for detecting buffer overflows and analyzing string library usage in C programs using a combination of memory modeling, static analysis, model checking and environment refinement. In contrast to other approaches, our approach uses carefully designed memory modeling along with a tight integration of scalable static analysis techniques (abstract interpretation) and SAT-based bounded model checking. However, limits on scalability of the underlying analysis make it hard to directly analyze industry-scale projects. In order to handle large projects, we partition the analysis problem into manageable parts, and use environment constraints to drive the analysis of these partitions. The environment constraints are refined in a demand-driven fashion, based on the violations that are deemed false. Thus, we are able to iteratively reason about the calling environment and guide the analysis towards fewer false positives and more non-trivial bugs.

We have implemented this approach in the tool F-SOFT to check for overflows in C programs. F-SOFT supports the checking of null termination of C strings and the usage of standard library functions. We present experimental results on using F-SOFT to analyze open-source as well as industrial projects.

Categories and Subject Descriptors: D.2.4 [Software/Program Verification]: Assertion Checkers; F.3.1 [Reasoning About Programs]: Pre- and post-conditions

General Terms: Verification, Reliability, Security

Additional Key Words and Phrases: Environment Refinement, Abstract Interpretation, Static Analysis, Model Checking, Buffer Overflows, F-Soft, Memory Model.

1. INTRODUCTION

Buffer overflows are common in systems code. A large number of overflows are present in deployed commercial as well as open-source software, presenting numerous problems with reliability and security [Cowan et al. 1999]. To complicate matters, many of the overflows (upto 70%) arise from the improper use of standard C library functions such as `strcat`, `strcpy`, `...`. A significant volume of

Address: NEC Laboratories America, 4 Independence Way, Princeton, NJ 08540.

Draft, December 2008.

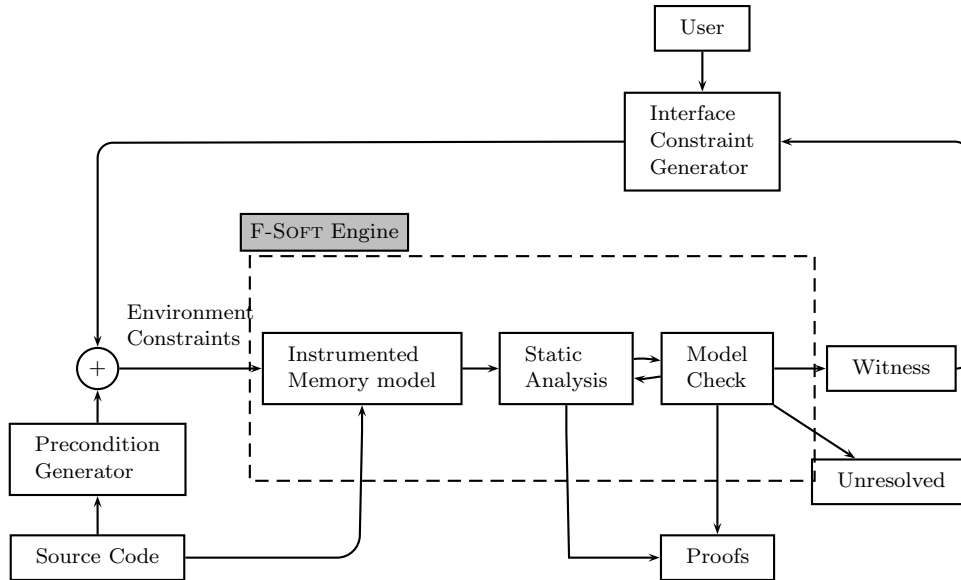


Fig. 1. Overall flow diagram for the CEGER framework using the F-SOFT analysis platform.

research on buffer overflow prevention has focused on the static detection of overflows. Tools that employ expensive analysis techniques are not scalable [Dor et al. 2003], whereas lightweight approaches suffer from too many false positives [Wagner et al. 2000; Bodik et al. 2000; Zitser et al. 2004]. In the latter case, the diagnosis of false positives requires significant human effort. Recently, approaches based on predicate abstraction such as SLAM and BLAST have been quite successful in analyzing “control-dominated” properties such as API usage for large C programs [Ball et al. 2001; Henzinger et al. 2002]. However, our own attempts at applying these techniques for checking memory safety properties have not been successful. Existing approaches may fail to be adequate in other respects. First of all, many tools do not provide adequate information about the alarms to aid such a diagnosis. Secondly, industrial-strength approaches need to systematically handle complications such as incomplete code (missing library functions) and incompletely specified calling environments. Finally, many existing approaches ignore or understate the importance of methodological concerns that arise while applying verification to large software.

In this paper, we present a comprehensive technique to detect buffer overflows in C programs using precise memory modeling, static analysis and model checking. More significantly, we describe an experimentally validated methodology based on counterexample guided environment refinement (CEGER) to systematically analyze large projects and provide precise counterexamples to property violations. The major contributions of this paper are:

- We present a precise *heap-aware* memory model for analyzing overflow properties of C programs. Our model is more sophisticated than related approaches such as CSSV [Dor et al. 2003]. It automatically adds instrumentation for tracking

array length, allocated sizes of pointers and sentinels for the null terminator character. It also handles pointer arithmetic, dynamic allocation, casting, multiple pointer indirections, arrays of pointers and standard library usage. Furthermore, it has been carefully designed to work well with numerical abstract interpretation techniques. Specifically, our design choices enable lightweight numerical abstract domain such as *intervals*, *octagons* and *symbolic ranges* to effectively handle the constructs used in our model.

- We employ a combination of static analyses using numerical domains for proving properties, and a bit-precise model checker for finding concrete bug traces for buffer overflow detection. Although some of these techniques have been used *individually* before, we present a tighter integration by staging the analyses and sharing information between them.
- In our tightly integrated framework, static analyses simplify the model considerably by reducing the number of variables, slicing irrelevant portions of the code based on proved properties, constant folding and restricting ranges for variables. The invariants computed using abstract interpretation are also used by the model checker to provide additional constraints during the search in model checking, leading to a significant performance improvement [Ganai and Gupta 2006].
- We propose a user-driven environment refinement method based on automatically generated counterexamples, called *CEGER*. Starting from some default environment constraints, this method iteratively weeds out false bugs due to inadequate environment modeling, in order to find more (true) bugs. The *CEGER* method also handles complications such as missing functions and unspecified calling environments. It provides API support for specifying stubs that serve as environment models. It also provides support for their refinement, by automatically suggesting a weakest pre-condition of the bug reported by the model checker.
- We present a technique to jump-start the process of refining the initial environment by a process of pattern-based precondition inference and *hoisting* preconditions up the call graph using data-flow analysis. The resulting techniques are implemented in the tool *SpecTackle* that is a part of the F-SOFT platform.

Our static analysis engine employs *abstract interpretation* [Cousot and Cousot 1977] to prove properties using numerical abstract domains such as *constants*, *intervals*, *octagons*, *symbolic ranges*, *polyhedra* [Cousot and Halbwachs 1978; Miné 2001a; Cousot and Cousot 1976; Sankaranarayanan et al. 2007]. We employ path sensitive extensions using CFG elaborations [Sankaranarayanan et al. 2006] and widening with care-sets [Wang et al. 2007].

We subsequently use model checkers, in particular bit-accurate SAT-based bounded model checking (BMC) [Biere et al. 1999; Ganai and Gupta 2006], to discover violations for the remaining properties. We also perform symbolic model checking using *mixed symbolic representations* [Yavuz-Kahveci et al. 2005; Yang et al. 2006]. Ultimately, all model checking engines output *concrete witnesses* (or counterexamples) that demonstrate (a) the path taken through the programs, and (b) concrete values for the program variables at these program points. The model checkers also generate a (*weakest*) *pre-condition* on the calling interface of the program, with respect to the concrete witness path.

Our techniques have been implemented inside the F-SOFT software verification platform. Our approach reliably handles individual problems of sizes in the order of 30 KLOC. However, industrial projects of interest are much larger: typically in the order of 100s of KLOC and beyond. Furthermore, they tend to be *incomplete*: we often analyze programs that rely on external libraries that are not available at the analysis time. Our approach relies on partitioning the verification problem and verifying each module in isolation. However, partitioning the problem can give rise to many false witnesses due to abstraction in the model and nondeterministic environments.

In order to handle the false positives that result from the partitioning, we use *environment constraints* at key interface functions. These constraints restrict the environment (input variables, unknown functions, etc.) at the start of the analysis. Therefore, the analysis ensures that the initial state satisfies the environment constraints. However, rather than require these annotations upfront, we start with an initial default environment constraint and permit the user to redefine these constraints in the presence of false bugs. Unlike approaches based on annotation checking, our tool is primarily oriented towards finding concrete violations of properties. Proofs are employed primarily to simplify the model, reduce the number of properties to be checked by the concrete model checking engine.

Since the default environment can produce false alarms, our approach supports the diagnosis of such alarms using the counterexample presented by the model checker. To avoid this violation in future runs, the user may redefine the existing environment constraints. New environment constraints are also discovered automatically using the weakest precondition and presented as a suggestion to the user. At each iteration, the verification is re-run using the refined environment constraints for the relevant partitions. The resulting witnesses are used to again refine the environment constraints, as needed. As a result of our analysis, the false witness rate decreases dramatically, while more concrete bugs are exposed. Our environment refinement technique is similar to the assume-guarantee reasoning used in tools such as annotation checkers. It is also similar to the automatic techniques used for predicate-abstraction refinement. However, the key differences are (a) we focus on environment refinement for reducing the false positive rate for bug detection, and (b) in contrast with predicate-abstraction refinement approaches, the model itself remains the same across environment refinements. Our approach seeks property proofs to indicate the lack of a bug, and to enable model simplifications. Finally, F-SOFT has been enhanced with new features and productized by developers inside NEC into a tool called VARVEL [Tokuoka et al. 2007]. VARVEL has been successfully used as a verification tool inside NEC .

Example 1.1. *Overflows related to string operations are quite common in software and hard to detect. Fig. 2 shows an example taken from the NIST SAMATE reference dataset that is classified as being error free [National Institute for Standards and Technology]. However, our analysis detects an overflow in the line labelled **L0** inside function “inputFiltering”. The overflow is realizable by providing an input argument which has a ‘/’ character in its 255th position and is of size at least 256.*

<pre> int main(int argc, char *argv[]){ if (argc > 1){ char fName[256] = ""; strcpy(fName, argv[1], 255); inputFiltering (fName); ... } } </pre>	<pre> void inputFiltering(char *fName){ char buf[256]="",*c = fName; char *b = buf; L0: for(; *c != 0; c++){ while(*c == '/') c++; *b++ = *c; } strcpy(fName, buf, 255); } </pre>
--	---

Fig. 2. Overflow bug found by F-SOFT inside an example classified as “OK” in the SAMATE database

2. MEMORY MODELING & INSTRUMENTATION

We model arrays, pointers, and C strings residing in the program stack and heap in a *compiler independent* manner. The memory model is incorporated by adding instrumentation to track pointer bounds, string lengths, and pointer validity. We also add assertions corresponding to pointer indirections and string library usage. The subsequent analyses that operate on the transformed code are oblivious to the memory model. They simply implement the standard semantics of the control/data-flow in the transformed program. We support two different memory models that use the same underlying analyses: (a) a fine-grained array bounds model to check for overflows and (b) a light-weight *validity* model that handles bugs such as `malloc` failures, double-free, free of statically allocated memory, escaped pointers and so on.

Structure Flattening. C programs typically have compound structures and unions, which may in turn contain compound structures inside them. We *flatten* these structures by expanding the fields of structures/unions and treating them as distinct variables. For instance, a structure variable x containing fields f_1, \dots, f_n yields the set of variables $\{x, x.f_1, x.f_2, \dots, x.f_n\}$. Further expansion is possible when some of the fields f_i are themselves structures. Similarly, pointers to structures are also expanded and the resulting variables $x \rightarrow f_i$ themselves treated as pointers of the appropriate type. However, for recursive structures (such as lists and trees) the flattening is terminated at some fixed user-specified depth. The field accesses beyond this depth are not modeled. Assignments to such fields are ignored and references to them return a nondeterministic value.¹

As a result of flattening, assignments between structures, structure pointers, structures passed as parameters or return values of functions are expanded to include the fields of the structure. This transformation can be costly for some programs but nevertheless ensures field sensitivity in all our analyses. The cost of this

¹This model generally introduces potential memory corruption errors whenever an access is made beyond the fixed flattening depth. To reduce the number of such false warnings, the user has the option to change the modeling such that all input recursive data structures are properly terminated using NULL values beyond the fixed flattening depth.

transformation is alleviated significantly by performing a flow-sensitive program slicing up-front to remove fields that are irrelevant to the properties being checked.

2.1 Array Bounds Checking

For checking array/pointer bounds, our memory model assigns positive integer addresses for the program variables and address ranges for statically allocated arrays. The address 0 is reserved for invalid memory addresses (for instance, the NULL pointer). Addresses are assigned using a pass through the program variables, assigning each base-type variable a unique address, and each static array a range of addresses depending on its size and the size of its elements.

Each pointer p is modeled by its integer address, and its bounds $[\text{ptrLo}(p), \text{ptrHi}(p)]$. The bound represents the range of legal values of the pointer p such that p may be dereferenced in our model, *without causing a memory error*. By convention, a pointer p is regarded invalid (uninitialized, null, freed, etc...) whenever $\text{ptrHi}(p) < \text{ptrLo}(p)$. The NULL pointer is treated as invalid.

We do not model memory at the byte-level. Addresses in our model *do not* represent the physical layout of the memory in some linear array. Providing an address to a pointer allows us to model pointer arithmetic and aliasing in an easier fashion.

Example 2.1. *A pointer `struct foo * p` in a program with attributes $p = 1000$, $\text{ptrLo}(p) = 910$, $\text{ptrHi}(p) = 1009$ indicates that $p[9], p[0], p[-10], p[-90], \dots$ are all valid accesses, whereas $p[-100], p[11], \dots$ are invalid.*

For statically allocated arrays, the bound variables and addresses are constants. Dynamically allocated pointers are not provided with a priori fixed addresses. Instead, `malloc` calls are instrumented to assign values for addresses and bounds for these pointers. The bounds for input pointers are nondeterministically selected according to the environment assumptions (see Section 4).

Our model tracks the values of locations corresponding to program variables and structure fields. However, for performance reasons, it does not track values of all the elements of an array. Instead, for each array a the model tracks the values of at most ℓ locations.

So far, we have assumed that the program does not contain recursive functions. In practice, recursive functions are seldom present in the commercial software that we have encountered. In the presence of recursion, we expand the recursive call to a fixed depth $f \geq 0$. The call at depth $f + 1$ is treated as nondeterministic. Each unrolled instance is treated as a distinct function call and the local variables are renamed to retain soundness. For the remainder of this paper, we assume that the code being analyzed is recursion free.

In order to ensure a scalable analysis of the model and at the same time finding useful bugs, our model departs from the usual C runtime model in the following significant ways:

- (a) *It blurs the distinction between addresses in the stack and the heap.* As a result, our model may access stack memory that may be out of scope. We address this by running an *escape analysis* prior to our analysis to ensure that such accesses are not present (see Section 2.3 for details).

<pre> void foo(int N) { /*ASSUME(0 < N < 100);*/ str_t ptr[100]; str_t * tmp; for(tmp=ptr ; tmp < ptr+N; ++tmp) initStrct(*tmp, ...); ... </pre>	<pre> void foo(int N) { ASSUME(N > 0 && N < 100); /* Magic number address */ ptr := 1012; ptrLo(ptr) := 1012; ptrHi(ptr) := 1012 + 24 *100; tmp := 1012; ptrHi(tmp) := 3412; for(; tmp < 1012 + 24 * N; tmp := tmp + 24) ASSERT(tmp ≤ ptrHi(tmp)); initStrct([*] , ...); ... </pre>	<pre> void foo(int N) { ASSUME(N > 0 && N < 100); /* Magic number address */ ptr := 1012; ptrLo(ptr) := 1012; ptrHi(ptr) := 1012 + 100; tmp := 1012; ptrHi(tmp) := 1112; for(;tmp < 1012 + N; tmp := tmp + 1) ASSERT(tmp ≤ ptrHi(tmp)); initStrct([*] , ...); ... </pre>
(a)	(b)	(c)

Fig. 3. (a) A program iterating over an array of structures, (b) word accurate memory model and (c) our memory model.

- (b) *It assigns contiguous addresses to locations that need not be placed contiguously by a real compiler or operating system, and vice-versa.* Even though we may assign two locations p, q consecutive addresses, we set the bound variables $\text{ptrHi}(p)$ and $\text{ptrLo}(p)$ so that any indirection of a pointer p outside the range $[\text{ptrLo}(p), \text{ptrHi}(p)]$ is considered a memory error.
- (c) *It need not place fields of a structure contiguously that are contiguous in most C compilers.* This limitation can be lifted by interleaving the fields of structures and using a stride to resolve pointer arithmetic. However, since compilers do not necessarily guarantee a fixed layout scheme for structure elements, we do not enforce any such layout in our memory model (Cf. Fig. 4). Consequently, our model does not handle *byte-level arithmetic* that assumes prior knowledge of the physical memory layout enforced by some compiler. Secondly, interleaving structure fields would introduce complex arithmetic expressions in our models. This would place a burden on our analyzer, significantly impacting the performance of static analyses such as *octagons*. With the exception of type-casting, our instrumentation produces *linear expressions* with *unit coefficients*, which are easy to reason with. Since pointer indexing occurs much more frequently in programs as compared to type-casting, we find this trade-off useful.
- (d) *It does not model array contents, recursive data structures and recursion completely.* While this is a limitation of our modeling, it can be alleviated, in part, by a user increasing the flattening depth for recursive structures, the modeling limit for array elements and the unrolling depth for recursive calls. However, increasing these parameters adversely impacts the performance, especially that of the model checker.

Our modeling choices have been made to enhance the power of lightweight analysis to reason about the program.

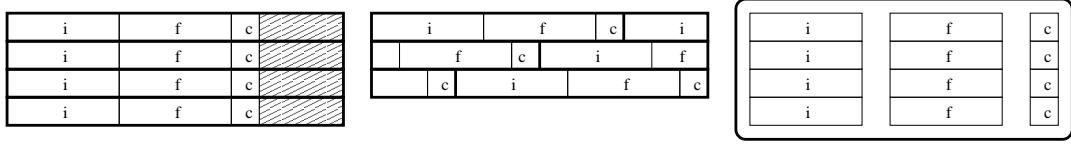


Fig. 4. Memory layout of an array of structure. **Left, middle:** Possible physical layouts by a C compiler, **right:** our memory model.

Example 2.2. *The program shown in Fig. 3(a) iterates through an array of structures, each structure having a size of 24 bytes. Fig. 3(b) shows a word-accurate model that treats each pointer as a 4 byte address, whereas Fig. 3(c) shows our memory model.*

Both the models shown are capable of proving the memory safety assertion $\text{tmp} \leq \text{ptrHi}(\text{tmp})$. However, the complexity of the invariants required are different in the two models.

The word accurate model requires the invariant

$$I_w : \text{tmp} < 1024 + 24 * N \wedge \text{ptrHi}(\text{tmp}) - 24 * N = 1024.$$

whereas our model requires the invariant

$$I_o : \text{tmp} < 1024 + N \wedge \text{ptrHi}(\text{tmp}) - N = 1024.$$

At the first glance, these invariants are almost identical, save for the multiplier 24 occurring in the former. The invariant I_o involves two variables with unit coefficients, and can be inferred for our memory model using an abstract domain with lower complexity such as octagons [Miné 2001a]. However, the invariant I_w requires more expensive invariant generation techniques such as polyhedral analysis [Cousot and Halbwachs 1978]. Whereas, octagon analysis can scale to large programs, we have had considerable difficulties obtaining a scalable and precise polyhedral analysis.

A word-inaccurate memory model such as ours allows for easy translation of pointer arithmetic. On the other hand, typecasts in a word-accurate memory model translate to direct assignment. Whereas, our model requires scaling by ratio of the element sizes to handle casting (Cf. subsequent discussion). However, since casting is much less frequent than pointer arithmetic, our model remains more amenable to static analysis.

Example 2.3. *Figure 4 illustrates our memory layout for an array of 4 elements, each of which is a structure with fields `int i`, `float f`, `char c`. The layout on the left shows the actual word-aligned layout. The layout in the middle may happen when the implementation of the structure is packed to avoid the padding between the structure elements. The C standard does not, however, specify a layout of the structure fields in the memory.*

The layout on the right, corresponds to our model. It dis-associates the structure fields from each other. Doing so, also simplifies the model construction considerably. However, we maintain the invariant that if a structure pointer is invalid, then

Assign $p := ?$	Instrumentation		
	New assign	$\text{ptrHi}(p) := ?$	$\text{ptrLo}(p) := ?$
$\text{malloc}(m)$	$p := \text{pos}$ $\text{pos}+ := m$	$p + m - 1$	p
$q + i$ & x	$p := q + i$ $p := c_0$	$\text{ptrHi}(q)$ c_1	$\text{ptrLo}(q)$ c_2

Table I. Instrumentation for basic pointer assignments. p, q denote pointers, x denotes a program variable and c denotes a constant.

pointers corresponding to the individual fields are all invalid. Likewise, we expand the *free* of a structure pointer to invalidate all its fields.

2.2 Model Construction

We now sketch the source code transformations that enable model construction. For the time being, we assume that all program variables and arrays have been provided address and ptrLo , ptrHi ranges according to their declared sizes. We now describe the instrumentation of dynamic allocation, assignments between pointers, pointer arithmetic, type-casting and pointer indirections.

Pointer Analysis. We first perform a pointer analysis to determine whether two pointers in the program may alias each other. The pointer analysis used is *flow-insensitive* (Steensgaard/Andersen) [Hind 2001]. It summarizes the points-to sets for arrays, dynamic allocation sites and recursive data-structures. Our analysis is implicitly field-sensitive due to the initial flattening. Pointer analysis is an important enabling step for model construction. Curiously, a more precise pointer analysis impacts the size of our model but not its semantics. In fact, unification-based flow-insensitive pointer analysis seems to suffice for the subsequent analyses.

Table I shows the instrumentation for the basic types of pointer assignments.

Dynamic Allocation. We use a special counter $\text{pos}(L)$ for each allocation site L in the code, to keep track of the maximum address currently allocated. Upon each call to a function such as $p := \text{malloc}(n)$, our model assigns the variable $\text{pos}(L)$ to p and sets bounds accordingly. It increments $\text{pos}(L)$ by n . Note that the memory modeling is not faithful to the actual semantics of the malloc call. For instance, it does not deal with de-allocation of memory fragments upon freeing.

Indirection. At each indirection $*p$, we add the assertion check $\text{ptrLo}(p) \leq p \leq \text{ptrHi}(p)$ to the model. If the assertion fails, the model sets $*p$ to a nondeterministic value. If the assertion succeeds, the points-to set of p is used to translate $*p$.

Let x_1, \dots, x_m be the (non-array) variables in the points-to set of p , and a_1, \dots, a_n be the arrays (or dynamic memory allocation sites). The value of $*p$ may evaluate to any of x_1, \dots, x_n or $a_i[p - \text{ptrLo}(p)]$. The values of program variables x_1, \dots, x_n are tracked in our model. However, not all array elements are tracked. Therefore, the value of $*p$ is evaluated using the following *if-then-else* expression shown below using C syntax:

$$*p : (p == \&x_1)?\mathbf{x}_1 : (p == \&x_2)?\mathbf{x}_2 : \dots : \text{ND}.$$

Note that if p is ascertained to point to an array, its value is treated as non-deterministic (ND). In our implementation, however, we track the values of a fixed number of indices for such arrays. The actual expression for such cases is not shown here. involves further case splits comparing the value of p with addresses of explicitly modeled array elements. If $*p$ is also a pointer, its `ptrHi` and `ptrLo` values can be obtained by a conditional expression based on its points-to set.

Likewise, an assignment of the form $*p := e$ is handled by considering the points-to set of p . If x_1, \dots, x_m belong to the points-to set, we expand the assignments to $*p$ in terms of assignment to x_1, \dots, x_m . The assignment to x_i is as follows:

$$x_i := (p == \&x_i)?e : x_i.$$

The conditional expression $p == \&x_i$ is significant. Less sophisticated static analysis techniques treat conditional expressions nondeterministically. However, our context-sensitive static analyses and significantly, the model checker can resolve these conditional expressions to simulate the effect of a more precise pointer analysis. Therefore, our model does not lose precision due to the coarseness of the pointer analysis. A better pointer analysis such as [Whaley and Lam 2004] reduces the size and number of these expressions, which can make the model checking more tractable.

Type Casting. Consider an assignment $p := (\text{newType}*) q$. To enable alias comparisons, we set the address of p to that of q . However, we re-evaluate the bounds `ptrLo(p)`, `ptrHi(p)` so that array accesses $p[i]$ are handled correctly.

Example 2.4. Consider an `int` pointer p with the address $p = 100$, and the ranges `ptrLo(p)`, `ptrHi(p)` : [100, 499]. This implies that $p[0], \dots, p[399]$ are all valid dereferences. Consider the typecast $q := (\text{char}*)p$. Since 4 characters can represent 1 integer, it should be possible to perform $q[0], \dots, q[1199]$ without errors. More precisely, the typecast is instrumented in this case by the statements $q := p$, `ptrLo(q) := ptrLo(p)` and `ptrHi(q) := ptrLo(p) + 4 * (ptrHi(p) - ptrLo(p) + 1) - 1`. Since types of the variables are available statically, we resolve any typecast in this manner.

The assignment $p := (\text{newType}*) q$ is translated to the assignments $p := q$ and `ptrLo(p) := ptrLo(q)` to model the address and lower bound, respectively. The upper bound `ptrHi(p)` is expanded based on the ratio of sizes of $*p$ and $*q$. More specifically, we assign `ptrHi(p)` to the following expression:

$$\text{ptrLo}(q) + (\text{sizeof}(*q)/\text{sizeof}(*p))(\text{ptrHi}(q) - \text{ptrLo}(q) + 1) - 1.$$

The ratio of sizes is a constant that can be determined statically, allowing for expression simplification.

In the case of down-casting (for instance, a structure pointer q cast as a `void` pointer p), the range `[ptrLo(p), ptrHi(p)]` subsumes the original range `[ptrLo(q), ptrHi(q)]` and may accidentally overlap with that of an unrelated pointer r that may potentially alias p according to the flow-insensitive pointer analysis. To avoid this, we introduce gaps in the address range assignments. The size of the gap is determined by the type casts that are being performed in the source code and the size of the arrays involved. Another issue is that the result of integer division may not be integral. To maintain soundness, we use the floor of the integer division.

Assignment $p := \langle \text{RHS} \rangle$	Instrumentation for $\text{ptrVal}(p)$
$p := \text{malloc}(\dots)$	$\text{ND?invalid} := \text{heap}$
$p := q + e$	$\text{ptrVal}(p) := \text{ptrVal}(q)$
$p := \&\text{globvar}$	$\text{ptrVal}(p) := \text{static}$
$p := \text{"const - string"}$	$\text{ptrVal}(p) := \text{code}$
$p := \&\text{locvar}$	$\text{ptrVal}(p) := \text{stack}$

Table II. Instrumentation of assignments for pointer validity check

<pre>int foo (int *q , int n) { int *p = q , i ; for (i = 0 ; i < n ; i++ , p++) if (*p > i) break ; return q[*p] ; }</pre>	<pre>int modelPV (int *q , FV ptrVal(q) , int n) { int *p = q , i ; FV ptrVal(p) = ptrVal(q) ; for (i = 0 ; i < n ; i++ , p++) { ASSERT(ptrVal(p)) ; if (*p > i) break ; } ASSERT(ptrVal(p)) ; ASSERT(ptrVal(q)) ; return q[*p] ; }</pre>	<pre>void simplePV (FV ptrVal(q)) { FV ptrVal(p) = ptrVal(q) ; ASSERT(ptrVal(p)) ; ASSERT(ptrVal(q)) ; }</pre>	<pre>void simpleAB (int *q , unsigned ptrLo(q) , unsigned ptrHi(q) , int n) { int *p = q , i ; unsigned ptrLo(p)=ptrLo(q) , ptrHi(p)=ptrHi(q) ; ASSERT(p >= ptrLo(p)) ; for (i = 0 ; i < n ; i++ , p++) { ASSERT (p <= ptrHi(p)) ; if (*p > i) break ; } ASSERT(ptrLo(p) <= p <= ptrHi(p)) ; ASSERT(ptrLo(q) <= q + *p <= ptrHi(q)) ; }</pre>
---	---	--	---

Fig. 5. Pointer validity checker for a small sample source code

Freeing Memory. A call to `free` on a pointer p invalidates the pointer as well as each pointer that aliases it. The aliased pointers are obtained from the alias set. Conditional expressions are used to ensure actual aliasing before pointers are invalidated. Note that it is not permissible in C to free a pointer p after performing some arithmetic on it. Let p_1, \dots, p_m be the aliases obtained through pointer analysis. The call to `free` invalidates each pointer p_i based on the condition $\text{ptrLo}(p) == \text{ptrLo}(p_i)$. Invalidating a pointer p simply involves assignments to the monitoring variables $\text{ptrLo}(p)$ and $\text{ptrHi}(p)$ such that $\text{ptrHi}(p) > \text{ptrLo}(p)$.

2.3 Pointer Validity Checking

The pointer validity checker handles those aspects of buffer overflow checking that do not include the tracking of bounds variables ptrHi , ptrLo for pointers. The validity checker instruments each pointer using a five-valued monitor $\text{ptrVal}(p)$ to denote the location of a pointer in the runtime memory organization: (a) **invalid**: an invalid pointer, whose dereference may cause a segmentation violation, (b) **static**: global variables, arrays and static variables, (c) **stack**: local variables, `alloca` calls, local arrays, formal arguments; (d) **heap**: dynamically allocated memory on the heap, and (e) **code**: code section, e.g., string constants.

Unlike the overflow checker, the validity checker does not track addresses of pointers. Validity of a pointer p simply suggests that the pointer p is sourced from an array that is currently in scope or dynamically allocated memory that has not yet been freed. The validity checker ignores address arithmetic. A pointer expression $p+i$ has the same validity status as its base pointer p . Table II shows the instrumentation for the pointer validity status. A dereference $*p$ yields an assertion check that is violated if $\text{ptrVal}(p)$ is invalid. In case of an assignment to $*p$, we may also report a bug if $\text{ptrVal}(p) = \text{code}$. Like the overflow checker, we use the alias set

Assignment $p := e$	RHS Translation $\text{strLen}(p) := ?$
$p := \text{malloc}(m)$	$\text{ND}_{>0}$
$p := \text{calloc}(m)$	0
$p := q + i$	$i \leq \text{strLen}(q) ? \text{strLen}(q) - i : \text{ND}_{>0}$
$p := \&x$	$(x == 0) ? 0 : \text{ND}_{>0}$

Table III. Instrumentation of assignments to track string lengths

of a pointer p to propagate the validity status across dereferences. Calls to `free` set the validity monitor of the argument and its aliases to `invalid`. Finally, leaving a functional scope changes pointers that are set to `stack` to have the validity monitor set to `invalid`.

Example 2.5. Figure 5 illustrates the pointer validity checker on a small but complex function `foo`. The figure shows the instrumentation of the function for pointer validity (`modelPV`) and its simplification (`simplePV`) by program transformations such as loop optimizations and abstract interpretation techniques that will be described subsequently in Section 3. It also shows the simplified array bounds instrumentation (`simpleAB`).

The pointer validity checker is thus able to find problems if the function is called with an initially invalid pointer q . Other problems, may exist, however. For instance, the indirection `*p` in the return statement can lead to a potential overflow. Such an overflow, however, needs instrumentation of the bounds `ptrLo(p)`, `ptrHi(p)`. It is revealed by checking the function `simpleAB`.

2.4 Tracking String Lengths

Conventionally, strings in C are represented as an array of characters followed by a special null termination symbol. String library functions such as `strcat`, `strlen`, and `strcpy` rely on their inputs to be properly null terminated and the allocated bounds to be large enough to contain the resulting strings.

Corresponding to each character pointer p , the variable `strLen(p)` tracks the position of the first null terminator character starting from p . The updates to string length can be derived along the same lines as those for the pointer bounds with the exception of assignments involving pointer indirections. Table III shows the handling of some of the simpler cases. The treatment of string pointer indirection also involves a conditional expression on the addresses and string lengths of aliased pointers.

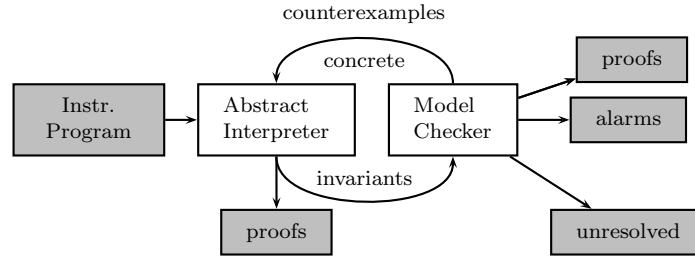


Fig. 6. Major analysis components in our framework

2.5 Standard Library Functions

Strings in C are manipulated using string library functions. The table below shows the preconditions for a few such functions:

function	Assertion checks
<code>strlen(p)</code>	<code>nullTerm(p)</code>
<code>strcpy(p, q)</code>	$\neg \text{overlap}(p, q) \wedge \text{nullTerm}(q)$ $\wedge \text{fwdBnds}(p) > \text{strLen}(q)$
<code>strcat(p, q)</code>	<code>nullTerm(p) \wedge nullTerm(q) \wedge</code> <code>fwdBnds(p) > strLen(q) + strLen(p)</code>
<code>memset(p, x, N)</code>	$(\text{ptrHi}(p) - p) * \text{sizeof}(*p) \geq N - 1$

All functions additionally require the input pointers to be well bounded, i.e. $p \in [\text{ptrLo}(p), \text{ptrHi}(p)]$. We use `fwdBnds(p)` to represent $\text{ptrHi}(p) - p + 1$. We write `nullTerm(p)` to denote the condition `fwdBnds(p) > strLen(p)`, which denotes that p is properly null terminated. The assignments shown below model the effects of the function calls:

function	<code>strLen(p) := ?</code>
<code>strcpy(p, q)</code>	<code>strLen(q)</code>
<code>strcat(p, q)</code>	<code>strLen(p) + strLen(q)</code>
<code>memset(p, x, N)</code>	$(x == 0) ? 0 : \text{ND}_{\geq N}$

3. ANALYSIS FRAMEWORK

Figure 6 shows the major analysis modules used in our framework. The overall flow is geared towards maximizing the number of property proofs and concrete witnesses to property violations. After model construction, we analyze the model using *abstract interpretation* over numerical domains with increasing degrees of precision. Each assertion proof indicates that the corresponding violation is not feasible in our model. Therefore, such properties can be removed from the model. We perform a backward slicing on the final model based on the checks that remain unproven. Constants are propagated to simplify the model considerably.

The model is then analyzed by a series of model-checking engines including SAT-based bounded model checking. Typically, the model checker runs for a fixed amount of time. At the end of the model checking, we obtain concrete traces that demonstrate property violations in the model. These violations are mapped back to the source code and displayed using a programming environment such as *Eclipse(tm)*. We briefly describe the major components:

Slicing & Constant Folding. We propagate constant assignments using a standard interprocedural data-flow analysis. Secondly, we slice the model to include statements and conditions that are pertinent to each assertion check. In practice, slicing and constant folding are quite efficient and drastically reduce the model size. Slicing also gets rid of structure fields that are impertinent to the property being checked, unused instrumentation for string lengths and array bounds and so on.

Abstract Interpreter. Abstract interpretation [Cousot and Cousot 1977] is used in our flow as the main *proof engine*. While the focus of our effort is on finding bugs, proofs are valuable since they indicate the absence of a bug *w.r.t our modeling assumptions*. Furthermore, proofs of properties enable semantic slicing and simplification of the program, further reducing its size and improving scalability. Our framework performs abstract interpretation over various numerical domains. Such domains track relations between integer variables in the program at each program location. An assertion violation $\text{ASSERT}(\varphi)$ is proved by our abstract interpreter if φ can be established as an invariant. Our abstract interpreter is inter-procedural, flow and context sensitive. It is built in a domain-independent and extensible fashion, allowing for various abstract domains. Currently, we have implementations of abstract domains such as *constants*, *intervals* [Cousot and Cousot 1976], *octagons* [Miné 2001b], *symbolic ranges* [Sankaranarayanan et al. 2007] and *polyhedra* [Cousot and Halbwachs 1978]. These domains are organized in increasing order of complexity.

To enhance the number of properties proved, we have incorporated a path-sensitive analysis technique based on transforming the CFG using the concept of *elaborations* [Sankaranarayanan et al. 2006], and by disjunctive polyhedral analysis, using a modified widening-upto operator [Wang et al. 2007]. While path-sensitive analyses can be quite expensive, we employ a progression of analysis techniques, each more powerful than the preceding analysis. Thus, heavyweight analysis techniques operate on programs that are simplified using the results of the preceding analysis.

The invariants computed at each program point are used to infer safe intervals for the program variables. The intervals thus computed restrict the domain of these variables and are utilized by the model checker. For instance, if the range for a variable x is $[0, 100]$, we may use 7 bits to represent x , as opposed to 32 bits that would have been required in the absence of such information. This reduces the number of variables in the SAT problem, typically a 50 – 80% reduction, and speeds up the model-checker considerably. Furthermore, the invariants are directly converted to SAT formulas and used to constrain the model checking search. These optimizations have led to a large reduction in the size of the models and the time taken to check them [Ganai and Gupta 2006].

Modeling vs. Analysis. Our choice of abstract domains for the abstract interpreter has also influenced that of the memory model. To begin with, the memory model avoids performing non linear arithmetic such as multiplication of variables with each other. Therefore, linear abstract domains work well on the model. Secondly, the memory layout chosen avoids the use of *strides* in pointer arithmetic (except for handling typecasting). This ensures that pointer updates such as $p := q + i$ are directly interpreted in our model without having to scale i by $\text{sizeof}(*p)$. Further-

more, the unorthodox memory layout transformation depicted in Figure 4, avoids the need for complex arithmetic upon field accesses on (arrays of) structures. Instead, such accesses directly translate into array accesses on the flattened structure. Such a modeling allows us to use domains such as *octagons* and *intervals* that do not model complex arithmetic involving non-unit coefficients.

Self Limitation Analysis. Our memory model tracks the sizes of arrays and pointers in a partially field sensitive manner. However, it does not track all values/attributes of array elements, or field accesses beyond the cutoff depth. These are modelled as non deterministic values.

In practice, their presence can lead to many false counterexamples that cannot be remedied by the environment refinement techniques presented later in Section 4. Therefore, we include a conservative dataflow analysis that identifies assertion checks which *must* (data) depend on these non-deterministic values. The removal of such checks reduces the number of false positives considerably.

Note that self limitation analysis does not prune properties that depend on a non-deterministic environment. It primarily removes properties that depend on array contents or recursive structures. Such properties cannot be resolved by our memory modelling, or proved using user-defined annotations.

Model Checker. The model checker creates a finite state machine representation of the program to find bugs or proofs for the remaining properties. Each integer variable is treated as a 32 bit entity, character variables as 8 bits and so on. However, the interval information provided by the abstract interpreter is used to reduce the number of bits significantly. As a result, we can now construct a symbolic model of the program, by *bit-blasting* the variables, the dataflow operations and the control flow structure using the range information. Note that by using bit-accurate representations of all (finite bitwidth) operators, we obtain a bit-accurate symbolic model for the program. We use bit-accurate representations of all operators, ensuring that arithmetic overflows are modeled faithfully. The model checker verifies the symbolic model for the reachability of the embedded assertion violation checks. We primarily use SAT-based *bounded model checking* (BMC) [Biere et al. 1999; Ganai and Gupta 2006]. This technique unrolls the program upto some depth $d > 0$ and searches for the presence of a bug at that depth by compilation into a SAT problem. The depth d is increased iteratively until a bug is found or resources run out. We have also implemented verification using *BDDs* [J.R. Burch et al. 1994] and *mixed symbolic representations* [Yang et al. 2006] to find proofs and violations for properties. Significantly, all the model checkers generate a counterexample trace that displays violations concretely. Such a trace vastly simplifies the user inspection and evaluation of the error.

Witness Slicing & Diagnosis. We implement a path-based slicing algorithm on the witness traces [Jhala and Majumdar 2005]. The algorithm prunes irrelevant statements from the reported witnesses and aids user comprehension. We use a weakest precondition on the witness trace starting from the assertion violation to aid in its diagnosis. Our tool provides interfaces to the `Eclipse(tm)` front-end and can demonstrate the trace through a debugger `gdb` to visualize the trace (see fig. 10).

Nondeterministic choices made in the witness trace are treated as special variables in this computation. Our tool automatically diagnoses the possible causes of the witness, based on the variables that appear in the weakest precondition. Figure 10 shows a typical trace displayed by our tool.

Performance & Effectiveness

We measure the effectiveness of our approach in terms of its scalability and number of real bugs found vs. the number of false positives. Our precise treatment of pointer and string operations guarantee that false positives due to modeling limitations are kept as small as possible. However, there are definite limits to the scalability of our analyses. As with other analyses, scalability depends on the nature of the program. There is, in general, a wide variability in performance, depending on the amount of string usage, the use of pointers, heap, the number of structure fields and so on. So far, we have run experiments cumulatively on more than 1MLOC of open source and industrial source code. Our analysis seems to reliably handle upto 30 KLOC at a time. It has performed well on some instances as large as 50 KLOC.

4. ENVIRONMENT CONSTRAINTS & REFINEMENT

Apart from scalability, verifiers for large projects (esp. commercial systems) need to work with incomplete code-bases, with calls to many library functions which are not included in the analysis. Furthermore, a typical industrial project is too large to be analyzed monolithically by our engines. Therefore, it is split into modules that are called using interface functions. In practice, the physical layout of the code inside directories and files provides simple heuristics for automatically partitioning large projects for verification. The verification of a module starts from each of its interface functions. Furthermore, while verifying a module, we treat functions belonging to other modules as missing library functions.

However, restricting verification to smaller modules leads to unspecified calling environment at the module entry and code with missing function calls. Our tool treats these limitations by allowing user-supplied annotations that specify the calling environments and summarize interface functions. Rather than require such annotations upfront, our tool supplies a set of reasonable default assumptions for the initial verification run in order to minimize the number of false positives while exposing inaccurate environment constraints. The user may then provide annotations for some parts of the code, *on demand* to avoid these false positives. Iterating this process leads to a dramatic reduction in the number of false positives and an increase in the number of real bugs exposed. We present a counterexample-guided *environment* refinement framework(CEGER) that enables the user to systematically arrive at the stubs for the missing functions and preconditions, starting from a well-specified initial environment. The rest of the section discusses various aspects of the CEGER methodology in detail.

Default Assumptions. We supply default environment preconditions and default function summaries based on assumptions about function behavior. These assumptions are inspired by common use scenarios. Table IV shows some of the default assumptions about variable values, pointer bounds, aliasing, and functions whose code is not included in the analysis.

int/char Var.	Nondeterministic Input
Pointer Aliasing	Input pointers are <i>not</i> aliased
Pointer Address	option A: NULL option B: $p > 0$.
Pointer Bounds	option A: Invalid option B: $\text{ptrLo}(p) \leq p \leq \text{ptrHi}(p)$
String Length	option A: NULL option B: nullTerm
Func. missing src.	ptr. args are set to nondet, globals are not changed

Table IV. Automatic default assumptions supplied by our framework.

```

int makeSentence( char * sent) {
1: int i;
2: char word[40];
3: if (sent == NULL) return -1;
4: memset(sent,0,1024 * sizeof(char));
5: for (i=0; i < 10; ++i) {
6:   getWord(word,40);
7:   if ( strlen(sent) + strlen(word) + 1 <= 1024 ) {
8:     if (strlen(sent) > 0)
9:       strcat(sent,“_”);
10:    strcat(sent,word);
   }
}

```

Fig. 7. String manipulations to append words to a sentence

Input pointers are nondeterministically set to be invalid, or to a valid pointer address and bounds. If p is chosen to be valid, then p is set to a unique non-NULL address, $[\text{ptrLo}(p), \text{ptrHi}(p)]$ forms a valid range, and furthermore, $p \in [\text{ptrLo}(p), \text{ptrHi}(p)]$. The allocated length of p is set to be positive but nondeterministic.

Note that these assumptions do not lead to a strictly sound treatment of the environment. It is always possible to call a function p with an invalid pointer that is not a null pointer. However, false witnesses caused by indirection of such pointers can potentially be caused by the indirection of a null pointer. In effect, the default assumptions help reduce the number of cases that need to be considered by the user in the understanding of the alarms produced by our analysis.

Example 4.1. *Figure 7 shows a code fragment inspired by one of the bugs discovered by F-SOFT in a commercial project. The function `makeSentence` strings together words obtained from the function `getWord` as long as the resulting word fits into the array `sent`. The `strcat` call in line 9 contains a subtle off-by-one buffer overflow triggered by the incorrect length computation in line 7. The `strlen(sent)` call in line 7 causes an overflow in the subsequent loop iteration.*

The program requires environment specifications for (a) the input status of the pointer `sent`; (b) definition for the function `getWord`; (c) the initial string lengths of the pointer `sent` and array `word`.

The default assumptions allow for two cases: (a) `sent` is a NULL pointer and

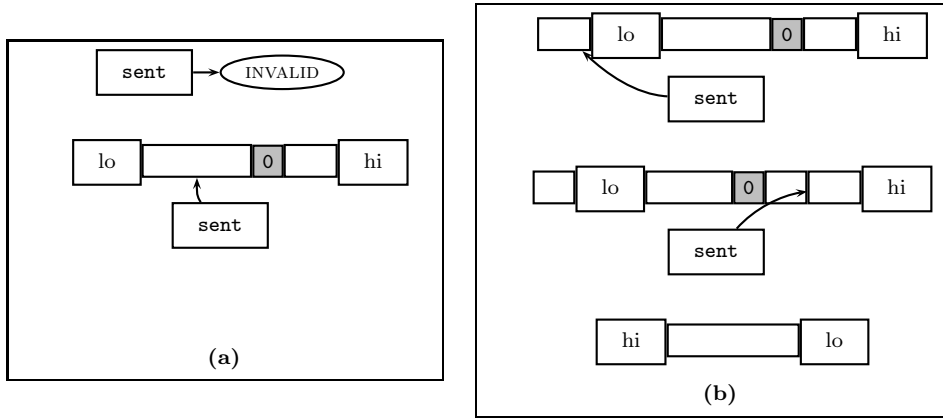


Fig. 8. Possible pointer configurations of an input pointer `sent`: (a) Part of default assumptions, and (b) not permitted by the default assumptions.

<code>preCond(expr)</code>	C expr. <code>expr</code> is a precondition.
<code>assert(expr)</code>	<code>expr</code> is asserted.
<code>assume(expr)</code>	<code>expr</code> is assumed.
<code>NONDET(e_1, e_2)</code>	A nondet. integer in $[e_1, e_2]$.
<code>setStrLen(p, e)</code>	Insert the assignment <code>strLen(p) := e.</code>
<code>setValidity(p, e)</code>	Assignment <code>ptrVal(p) := e.</code>

Table V. Stub API to specify environment

(b) `sent` has valid but nondeterministic array bounds. The function `getWord` is assumed to set the string length of `word` to an arbitrary value. As a result, we obtain violations to all the library function calls (lines 4, 7, 8, 9 and 10 of Fig. 7).

Fig. 8 depicts some of the cases that are considered by the default assumption for the input pointer `sent` along with some of the possibilities that are not considered by the default assumptions.

An examination of the violations suggests that (a) `sent` is expected to have a forward bound of at least 1024 and (b) `getWord` needs to ensure that the array `word` is properly null terminated. We allow the user to specify this by writing a stub for this function using an in-built API.

Stub API. The default assumptions may still lead to a large number of false positives. In many cases, removing false violations requires user input in the form of environment preconditions and function summaries. To aid the user in this process, we support a special *Stub API* that allows the user to directly write assertions, assumptions, preconditions and assignments involving instrumentation variables such as `ptrHi(p)`, `ptrLo(p)`, `strLen(p)`, `ptrVal(p)`, and so on.

Table V shows some of the constructs available to the user to write preconditions and summaries for missing functions. It allows us the user to directly control the model built. We interpret functions to specify preconditions, assumptions and assertions. The `assert` function is part of the C library. It is treated as a check. The

```

int _my_stub_4_getWord (char * ptr, int n) {
    int x = NONDET(0,n-1);
    setStringLength(ptr,x);
    return x;
}
void _pre_cond_4_getSentence (char * sent) {
    /**@ preCond( fwdBnds(sent) >= 1024 ); */
}

```

Fig. 9. A stub summarizing the behavior of function `getWord` and a precondition for the function `getSentence`.

assume function blocks the execution unless the assumed expression is valid. The precondition function `preCond` has two different meanings depending on where it is encountered in the analysis. It is used as an assumption when it is encountered at the entry function of the analysis. If encountered in a non-entry function, it is treated as a check. This corresponds to the natural semantics of a precondition in annotation checking. Postconditions are treated as assumptions that are enforced when the function is called from a different module.

Example 4.2. *Fig. 9 shows the stub function for modeling the behavior of the function `getWord` and the precondition for the argument `sent` to the function `getSentence` in Fig. 7, using a convenient mechanism provided.*

When used in our verification framework, it exposes the real off-by-one error in the program in line 10, corresponding to the `strcat` call. Our approach also detects the violation of the `strlen` call in line 7.

4.1 CEGER

The basic idea behind CEGER is to first perform analysis using the default assumptions. Afterwards, the user examines the initial set of witnesses and provides preconditions and stubs for missing functions as required. The stubs and preconditions are refined iteratively to avoid some of the false witnesses encountered in the previous iterations.

While the user is central to the analysis of the witnesses produced, a concrete witness trace allows automatic precondition-based refinement for traces that are deemed implausible due to a missing environment constraint. For the remainder of this section we assume that all the missing functions are handled using default assumptions or user-supplied stubs.

Our methodology focusses on the *refinement* of the calling environment for each function. Let φ_f be the existing environment precondition corresponding to an interface function f in the code. For each witness w , let ψ_w be the weakest precondition computed at the interface starting from the assertion violation. There are two possible diagnoses corresponding to a false witness w raised:

- (1) Starting from function f , we observe an assertion violation in the code that is dependent on the input environment. In this case, φ_f is too weak. We term such witnesses as *Type 1* witnesses.
- (2) Starting from some other function g , we violate the precondition φ_f (treated as an assertion) at one or more call sites to f . In this case, either φ_f is too

strong or φ_g is too weak. Witnesses of this form are termed *Type 2* witnesses.

In order to eliminate a *Type 1* witness w , we refine the precondition for the entry function f by conjoining it with $\neg\psi_w$. This suffices to eliminate w from future iterations of the framework. In many cases, this also rules out other related witnesses to the same assertion from alternative program paths. Note, however, that φ'_f is presented to the user as a suggestion. In many cases, the user may be able to rule out additional false witnesses by weakening φ'_f .

Example 4.3. Consider the initial run of the program in Fig. 7 leading to a violation at the *memset* call on line 4. This is a *Type-1* violation. We compute the weakest precondition: $\text{sent} \neq \text{NULL} \wedge \text{ptrHi}(\text{sent}) - \text{sent} < 1024$. Therefore, it automatically suggest the precondition:

$$\text{sent} == \text{NULL} \parallel \text{ptrHi}(\text{sent}) - \text{sent} \geq 1024.$$

To handle a *Type 2* witness, we re-run the analysis starting from function g and including the code for function f . If the analysis succeeds in producing a violation inside function f , its precondition is used to refine φ_g , as a *Type-1* witness. If the analysis fails, the user may choose to treat the original witness w as a *Type-1* witness to refine φ_g , or to relax the precondition of f such that $\llbracket \varphi'_f \rrbracket \supseteq \llbracket \varphi_f \rrbracket \cup s$, where s is the concrete state in the witness causing the violation of φ_f . Such a relaxation will also eliminate the witness w from future iterations of our framework.

It may however, be possible that the precondition for f may be too restrictive, and thus hide a plausible bug inside f . We can, however, diagnose the probable cause for a *Type 2* witness. Let s be the state of the program at the call site to f . Let $\varphi'_f \supseteq \varphi_f$ be an assertion that generalized φ_f to include s . We now analyze f with the precondition φ'_f to check if a *Type 1* error is possible. If not, we treat the witness w as a *Type 1* error for the function g and refine its precondition using the weakest precondition of w starting from the failed call to f . However, if a *Type 1* witness is possible in f , let φ'_f be the refined precondition of f w.r.t this error. Furthermore, if $s \in \llbracket \varphi'_f \rrbracket$ we also refine the precondition φ_g using the witness w to the call site of f .

Let φ_f, φ_g be the precondition assertions for f, g prior to the witness w and φ'_f, φ'_g be the preconditions obtained by our refinement process. It is possible to prove that φ'_f, φ'_g eliminate the witnesses starting from g and f from consideration.

The technique of restricting and relaxing the preconditions based on false witnesses may not converge, especially in programs containing loops and operating on unbounded data. However, we have observed empirically even one or two iterations of this process vastly decreases the number of false witnesses, while preserving the true bugs for the most part.

Finally, since the possibility of a witness being false is arbitrated by the user, mistakes by the user can lead to missed bugs. In particular, the user may use fallacious environment assumption, $\varphi_f : \text{false}$, proving all the properties trivially. The static analyses can detect such situations easily and warn the user.

Example 4.4. We illustrate our approach to find a well-known bug in the open-source program `bc-v1.06` [Lu et al. 2005]. Figure 11 shows the code relevant to the bug. The global pointer `arrays` has its allocated length stored in `a_count`. However,

```

Witness shows: Property in file storage.c at line 172: Trying to read or write a value via an
invalid pointer.

Weakest preconditions:

1. arrays <= 0
2. 1 < a_count

00160 /* Save the old values. */
00161 old_count = a_count;
00162 old_ary = arrays;
00163 old_names = a_names;
00164
00165 /* Increment by a fixed amount and allocate. */
00166 a_count += STORE_INCR;
00167 arrays = (bc_var_array **) bc_malloc (a_count*sizeof(bc_var_array *));
00168 a_names = (char **) bc_malloc (a_count*sizeof(char *));
00169
00171 for (indx = 1; indx < old_count; indx++)
00172 arrays[indx] = old_ary[indx];

```

Fig. 10. Snapshot of HTML witness for bc-v1.06 bug

```

void more_arrays() {/* storage.c, line 152 */
1:  old_count = a_count;
2:  old_ary = arrays;
3:  a_count += STORE_INCR;
   ...
4:  arrays = (...) bc_malloc(a_count*...);
   ...
5:  for (indx=1; indx < old_count; indx++) arrays[indx] = old_ary[indx];
   ...
6:  for (; indx < v_count; indx++) arrays[indx] = NULL;
   ...

```

Fig. 11. An overflow bug inside bc-1.06.

due to a copy-and-paste bug, the unrelated variable `v_count` is used as the loop bound in line 7.

An initial run of our pointer validity checker yields false witnesses at lines 4, 5, 6, which assume that the pointer `arrays` may be an invalid array, or not take the relationship between `arrays` and `a_count` into consideration. Figure 10 shows one such witness displayed by our tool (corresponding to label 5 in the figure) and the precondition computed this particular violation. Adding the negation of the suggested precondition rules out further witnesses for the validity checker. However, the array bounds checker using this precondition yields a false witness at the same location due to missing relation between the length of `arrays` and the variable `a_count`. In order to rule this witness out, we add the appropriate precondition. After providing this precondition, the array bounds checker discovers the real buffer overflow. The original false witness that inspired the precondition is no longer present. The combined analysis times for all stages in this example was under a minute.

5. AUTOMATED ENVIRONMENT GENERATION

The CEGER framework provides *default* assumptions on the environment at the entry function for the source code being verified. These assumptions are not *conservative*. However, they are sufficient to allow for important situations that may

```

void f0()                void f1( int * p)          void f10( int * p)
  int * p = malloc(10);  f2(p);                ...          *p = 10;
  f1(p);

```

Fig. 12. Example requiring repeated CEGER iterations to detect the null pointer access in `f10`.

lead to bugs, in practice. At the same time, they are *permissive* enough to avoid the large number of false positives that any completely sound model will necessarily bring about.

The default assumptions consider the case wherein an input pointer may be invalid (NULL), unless ruled out using a precondition. In practice, many false warnings are produced by wrongly assuming that a set of pointers may be null. To eliminate such warnings, the user is called upon to insert appropriate preconditions to the entry function and re-analyze the entire program using F-SOFT. In practice, pointers are frequently passed as arguments across function calls. As a result, the precondition $p \neq \text{NULL}$ may need to be propagated to other functions in the call graph. This may lead to many iterations of environment refinement, in practice.

Example 5.1. *Fig. 12 shows a simple program with n functions f_0, f_1, \dots, f_n wherein the function f_0 passes a potentially invalid pointer to f_1 (not checking the result of a `malloc` for NULL). Each function f_i passes its arguments directly to the subsequent function f_{i+1} . The leaf function f_n dereferences its argument. Suppose we run CEGER on each function in the code, separately. This will require n iterations to confirm the actual violation. The first iteration will compute the precondition $p \neq \text{NULL}$ for f_n . Each iteration then propagates this precondition one level higher in the call graph until it is propagated to f_1 and identified as a true bug.*

The call graphs for large programs are deep in practice. It is common to allocate memory at a higher-level function in the call graph, which may then be passed as arguments through many levels to lower-level functions. To alleviate this problem, we have implemented static analysis techniques to infer preconditions in a tool called `SpecTackle`. `SpecTackle` infers preconditions of a pre-defined fixed form such as $p \neq \text{NULL}$. It can also *hoist* the preconditions inferred by the analysis across the call graph to avoid the problem of repeated refinements.

The process of precondition inference consists of two parts: (a) *Infer* preconditions based by identifying common pointer usage patterns inside a function, and (b) *Hoist* preconditions across function calls in the call graph.

5.1 Pattern-based Inference

The inference of preconditions are based on pre-defined pointer usage patterns. Figure 13 depicts some of these usage patterns and the precondition inferred based on these patterns. Fig 13(a) shows the common case wherein a pointer q is dereferenced after adding an offset represented by the expression e . Corresponding to this case, we generate two annotations, one to ensure the safety of the access and the other to check for the common case wherein q may be a NULL pointer. Fig 13(b) extends this pattern to the common case wherein a pointer is accessed through the index variable of a for-loop such that the variables in the bound expression are not

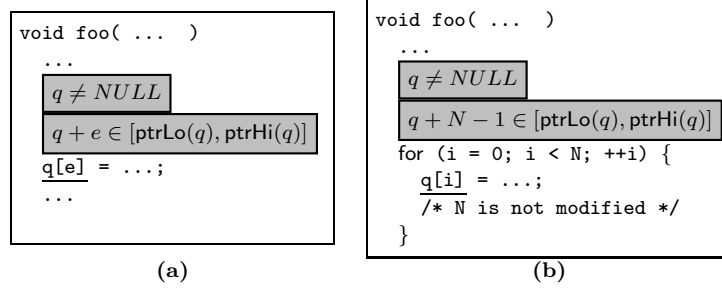


Fig. 13. Pointer usage patterns and the corresponding annotations generated locally at the usage point.

changed inside the loop. In this case, we add an annotation corresponding to the maximum address dereferenced inside the loop.

Note that the first stage is directed towards simple and well defined usage scenarios by means of a simple syntactic pattern matching. The locally generated annotation is then *hoisted* to the beginning of the function and further up in the call graph using a conservative expression pre-condition analysis.

5.2 Hoisting Annotations

Let $\varphi_l[X]$ be a pre-condition annotation generated at a program point l inside a function f involving variables in the set X . Our goal is to compute an appropriate precondition ψ corresponding to the entry point to the function f . This is computed by means of a backwards data-flow analysis that substitutes the effect of various assignments and conditional branches into the precondition.

Figure 14 shows the transfer functions for various types of assignments and conditional branches in the program. A direct assignment of a variable to an expression is handled using a substitution. Indirect assignments through potentially aliased pointers are handled conservatively by means of an over-approximate *projection* operator that eliminates occurrences of the aliased variables from the assertion. This is done by simply replacing atomic formulae involving these variables by *true* or *false*, depending on the structure of the formula. The possible side-effects of function calls can be handled similar to indirect assignments.

Finally, branches are handled using special meet/join operators in order to generate a correct pre-condition for the common cases and deal with issues such as the eventual termination of the analysis. The meet operator involves the precondition annotation being hoisted and a loop condition. Computing the meet operation involves the syntactic search for a conjunct in φ that contradicts the branch condition c . If such a conjunct is obtained, the meet is the assertion *false*. If, on the other hand, φ contains a conjunct that is identical to c (syntactically), the result of the meet is given the assertion φ itself. Formally,

$$\varphi \sqcap c = \begin{cases} \text{false} & \text{if } \varphi \text{ "syntactically contradicts" } c \\ \text{true} & \text{if } \varphi \text{ "is identical to" } c \\ \varphi & \text{otherwise} \end{cases}$$

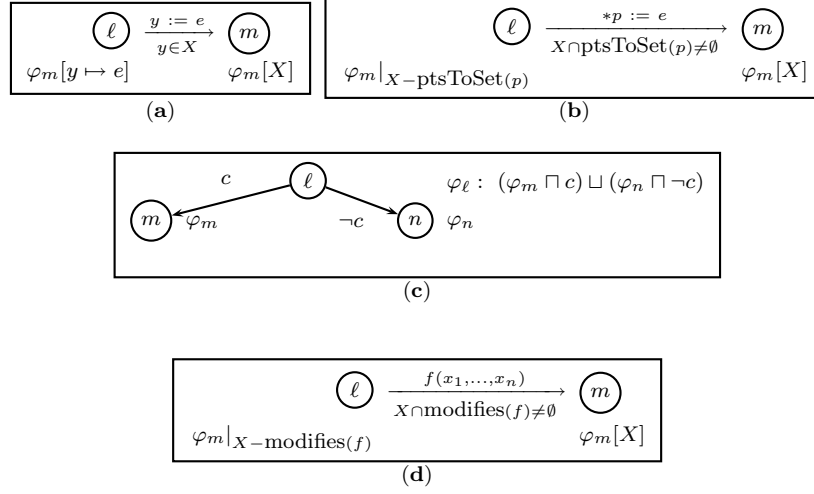


Fig. 14. Flow analysis for hoisting precondition annotations. (a) handling simple assignments, (b) aliased assignments, (c) conditional branches and (d) procedure calls with side effects.

A simple join operator is defined as follows:

$$\varphi \sqcup \psi = \begin{cases} \varphi & \text{if } \psi \text{ is syntactically false} \\ \varphi & (\psi \equiv \varphi \wedge \psi') \\ \text{true} & \text{otherwise} \end{cases}$$

The join operator works by syntactic matching of its operands. If one of the operand syntactically matches *false* then the result is taken to be the other operand. If ψ can be obtained by conjoining some assertion ψ' to φ then the join chooses the weaker assertion φ . Finally, if the operands do not fall into any of the categories above, the result of the join is the trivial annotation *true*.

The hoisting of an annotation φ_l at location l is initialized by labelling location l with φ_l and every other location by the assertion *false*. After the analysis converges, the assertion labelling the entry to the function denotes the entry precondition. This can be hoisted to the callers of the function if the precondition is an assertion other than *true*, *false*. If the formula *false* is produced, then a warning may be issued to the user.

Example 5.2. Consider functions f_1 and f_2 shown in Figs. 15(a & b), respectively. The pointer dereference $q[4]$ in line 3 of f_1 gives rise to two annotations φ_3 and ψ_3 respectively. Consider assertion $\varphi_3 : q \neq \text{NULL}$. Being identical to the condition $c : q! = \text{NULL}$ at line 2, the meet $\varphi_3 \sqcap c \equiv \text{true}$. The assertion φ_4 labelling line 4 is initially false. Joining the contribution across the two branches at line 2, yields $\varphi_2 : \text{true}$. As a result, the annotation $q \neq \text{NULL}$ does not yield a precondition.

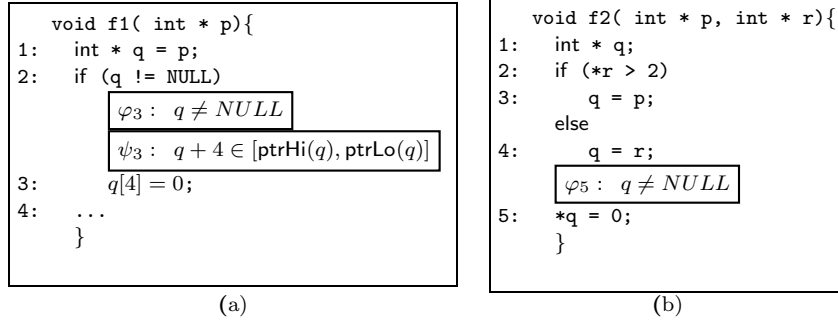


Fig. 15. Example functions manipulating pointers to demonstrate annotation hoisting.

On the other hand, hoisting the annotation $\psi_3 : q + 4 \in [\text{ptrLo}(q), \text{ptrHi}(q)]$ produces the precondition $p + 4 \in [\text{ptrLo}(p), \text{ptrHi}(p)]$.

Consider the annotation φ_5 in function $f2$. The annotation can be hoisted across the assertions in lines 3 and 4 yields $\varphi_3 : p \neq \text{NULL}$ and $\varphi_4 : r \neq \text{NULL}$, respectively. The join operation at the branch in line 2, yields the assertion $\varphi_2 : \text{true}$. As a result, no precondition is generated corresponding to the indirection.

In practice, a conservative precondition is not strictly necessary. For instance, an overly restrictive precondition annotation may lead to its violation at some call site. Similarly, too over-approximate a precondition may cause false violations based on an under-specified environment. In practice, our implementation of `SpectTackle` sacrifices soundness in its handling of pointer indirections. Nevertheless, the number of unsound preconditions generated is very few, in practice. Furthermore, all such instances can be detected through implausible witnesses produced by the model checker.

6. EXPERIMENTS

We ran experiments on some public benchmarks put forth by Zitser et al. to evaluate the performance of academic and commercial static analysis tools [Zitser et al. 2004]. The benchmark suite consists of 25 programs each of which is a part of a larger open source application with known overflow bugs involving arrays and strings. Significantly, each benchmark example initializes its environment in a manner that enables us to expose the underlying real bug or evaluate its fix. Therefore, the use of `SpectTackle` for precondition-based refinement was unnecessary for these benchmarks. Nevertheless, refinement was necessary for the false witnesses due to missing definitions for external functions. We allowed each instance to run for 1800s. The programs themselves range in size from 1–5kLOC (after pre-processing, forward slicing and instrumentation). Table VI shows the results of running the analysis on these examples. We are able to prove a majority ($\sim 65\%$) of the properties statically. In doing so, we also significantly reduce the model size (number of blocks and the number of variables). The model checker produces 90 concrete violations. Each violation was manually examined by one of the authors. Of the 90 witnesses found, 7 were found to be real violations based on the witness. The remaining 83 were classified into 4 categories. Table VII shows an analysis of these

kLOC			Avg. Model Size before abs. int.		Avg. Model Size after abs. int.		Avg. Time Seconds	
Tot	Avg	Max	#Blks	#Vars	#Blks	#Vars	AI	MC
46	1.9	4.5	1440	2546	245	93	555	1330

#Prop.	Proofs		Wit.	T/O
Total	AbsInt	BMC		
3281	2155	4	90	500

Table VI. Data from initial run of the Zitser et al. Benchmarks. **Legend:** AI: Abstract Interpretation; MC: Model Checking, Pr.: Proofs, Wit.: Witnesses, T/O: Time Out.

Status	# Wit (1)	#Wit. (2)
Plausible Bugs	7	10
Missing function <code>getopt</code> , <code>optarg</code> , ...	25	0
Missing function <code>dn_skipname</code>	12	0
Missing functions <code>setpwent</code> , <code>getpwent</code>	43	0
Modeling limitation: Array/string elements	3	3

Table VII. Analysis of witnesses from Zitser et al. Benchmarks

violations($\#Wit(1)$). Note that missing preconditions are not a problem for these benchmarks, since they contain drivers that initialize the environment.

Our subsequent iteration included a stub for the missing functions. The behavior of these functions were obtained by consulting their manuals. On the other hand, since we could not find references to `dn_skipname`, we arrived at an initial stub by simply examining the weakest precondition of the counterexample trace supplied by our tool. Table VII shows the results of the second iteration ($\#Wit(2)$). This iteration results in a total of 13 witnesses, of which 10 were found plausible. All in all, about one person hour was spent analyzing all the witnesses and deriving the stubs.

6.1 H-264 Video Decoder

We apply verification based on the CEGER approach to a medium sized and complex H.264 video decoder software written in C, consisting of 25,000 simplified lines of code after preprocessing. A detailed description of the decoder is available elsewhere [Nishihara et al. 2008]. The verification effort focussed on the serial (sequential) version of the decoder. Verification was attempted for 40 functions chosen based on the size of the source code exercised by each function. The size ranges from 2 – 15KLOC for these functions.

The decoder performs a complex set of bit operations on an input video stream. To enhance performance, the core bit-manipulation routines are implemented directly in assembly language. In order to jump-start the verification process, it was necessary to understand these routines and replace them by means of over-approximate stubs.

Automatic precondition inference was performed using the `SpecTackle` tool. The

Tot. Prec.	% Ptr. deref.	% Static Array	% Pointer indexing.
1473	69%	22%	9%

Table VIII. Preconditions generated for H.264 example

	SLOC Avg.	Tot. time (sec.)		Properties			
		Static	BMC	#Tot.	%Pr.	%War	T/M. O.
without prec.	2190	12617	152573	7237	64	9.3% (658)	26.7
with prec.	2671	20828	96382	12526	78.1	.3% (30)	21.6

Table IX. Performance of F-SOFT without and with automatically generated preconditions on H.264 example. Legend. SLOC: simplified lines of code, **Tot.:** total, **%Pr.:** percentage of properties proved, **%War.:** percentage of witnesses, **T/M.O.:** percentage of inconclusives due to time/memory out.

Total Witnesses	30
Plausible overflow bugs	10
Array element modeling	8
Missing preconditions	12

Table X. Analysis of warnings generated from H.264 witnesses (run with preconditions)

SpecTackle tool runs in a matter of seconds, generating nearly 1500 preconditions for all the functions in the source code. The types of these preconditions are summarized in Table VIII. Overall, a vast majority (69%) of the generated preconditions corresponded to direct pointer dereferences of the form $*p$, without any indexing. A significant number (22%) correspond to preconditions generated due to indexing of statically allocated arrays with known sizes, while the rest correspond to indexing of pointers of unknown sizes.

Automatically generated preconditions serve to jump-start the CEGER process. We ran F-SOFT with and without pre-conditions in order to perform a comparison. The overall performance of F-SOFT with and without automatically generated environment assertions is compared in Table IX. Note that the presence of preconditions in the code increases the average number of lines in the simplified source code that is input to our analysis. In general, the number of properties to be analyzed also increases in the presence of annotations. However, the presence of annotations enables us to prove a significantly larger percentage of properties. More significantly, the number of witnesses is reduced from a large number 658 without preconditions to a tiny fraction (30 witnesses) of the original witnesses, in the presence of these automatically generated preconditions.

Table X presents an analysis of the witness. It reveals 10 plausible buffer overflow bugs. These bugs are most likely caused due to the absence of a check on an array index into a large static data table in the heap. The index itself is obtained from a series of bit manipulations making bit precise modelling absolutely necessary for the detection of these witnesses. The remainder of the witnesses are caused due to limitations in the bounded array modeling of F-SOFT and the need for stronger (sometimes non linear) environmental preconditions.

7. RELATED WORK

Buffer overflows can cause memory corruption which may be hard to detect instantly. Cowan et al. survey different buffer overflow attacks and some attempts at prevention and detection [Cowan et al. 1999]. An overwhelming majority of approaches to buffer overflow detection use runtime analysis. However, such approaches exhibit a significant overhead, that may be unacceptable in some cases. As a result, there has been much interest in static detection of buffer overflows to detect potential vulnerabilities and to reduce the runtime overhead.

Static approaches use pointer analysis, range analysis and constraint solvers at various degrees of precision. Some approaches have been successful in eliminating a majority (40 – 60%) of array bounds checks in Java and C programs. Wagner et al. transform the overflow check elimination problem into one of solving interval constraints over integers [Wagner et al. 2000]. Bodik et al. use constraints generated from the SSA form to automatically reason about array bounds check [Bodik et al. 2000]. Rugina and Rinard provide a powerful summary-based approach that reduces interval analysis problems into linear programming [Rugina and Rinard 2000]. Along similar lines, Ganapathy et al. reduce overflow detection problems to linear programming to find likely overflows in systems code [Ganapathy et al. 2003]. Tools such as CCured integrate checks for pointers with automatic instrumentation to enable safe execution of low-level code [Necula et al. 2002]. However, scalable approaches do not completely handle complications involving dynamic memory allocation, strings, heap data-structures, array contents, standard library functions and type-casting. Recently, however, there has been work on more comprehensive approaches to overflow detection, that handle many of the complications mentioned above [Simon and King 2002; Dor et al. 2003; Christensen et al. 2003; Allamigeon et al. 2006].

The CSSV tool due to Dor et al. [Dor et al. 2003] approach constructs a memory model that tracks pointer bounds, and string lengths of arrays. It can check user-provided annotations intraprocedurally. A precise region-based points-to analysis is used to handle overlaps between strings. Our memory model is similar to that of CSSV. However, our instrumentation relies less on the precision of the pointer analysis. More complex analyses, including the model checker, can resolve these aliases naturally, based on the context and the invariants computed over pointer addresses. By combining abstract interpretation with SAT-based model checking, we obtain scalable analysis for programs that are much larger than those reported by Dor et al. The CSSV tool performs annotation checking and guarantees that no overflow is missed by the tool w.r.t the user provided precondition and postcondition annotations. In contrast, our tool supports a better environment refinement based on the concrete (bit-accurate) witness found by the model checker. This significantly reduces the burden on the user to supply environment annotations.

The HAVOC tool uses a low-level memory model to perform annotation checking on low level C programs [Chatterjee et al. 2007]. The memory model in HAVOC is quite detailed and like ours, it supports low-level casting, pointer arithmetic and dynamic allocation. However, to our knowledge the model does not support C strings. The lack of optimistic default assumptions necessitates a potentially large annotation burden on the user of the tool. Another potential drawback of

HAVOC and related approaches is that the byte-level modeling of memory makes the invariants required for verifying common overflow properties fall outside the purview of simple abstract domains such as octagons (recall Ex. 2.2).

Our approach, on the other hand, is geared towards finding useful bugs. Therefore, we tolerate some degree of unsoundness that may lead to missed overflows. Furthermore, the user annotations are provided on demand to refine the environment assumptions to avoid false positives. In practice, we find that a mathematically sound treatment of complications such as missing code for functions, and unmodelled environments leads to numerous overflow alarms that make the tool unusable without significant user interaction. We make some *default* assumptions on these cases, that the user can override. Finally, our approach uses static simplifications to aggressively simplify the program. Based on published data, our tool can analyze program fragments that are much larger than CSSV or HAVOC.

Our approach uses the theory of abstract interpretation due to Cousot & Cousot [Cousot and Cousot 1977] along with numerical domains such as Intervals [Cousot and Cousot 1976], Octagons [Miné 2001a], Polyhedra [Cousot and Halbwachs 1978] and other numerical domains of intermediate precision and complexity. Approaches based on abstract interpretation have been used in tools such as *PolySpace* [pol], *Astrée* [Blanchet et al. 2003], *C Global Surveyor* [Venet and Brat 2004], *Clousot* [Barnett et al. 2008] and so on. With the exception of *Clousot*, these tools focus on checking embedded applications with special features such as known aliasing patterns, no dynamic allocation, regular control flow and no recursion. However, our tool is designed to be more general purpose. Also, to our knowledge, these tools do not check null termination for strings. Furthermore, the emphasis on total soundness of proofs requires a combination of complex abstract domains, refinement, and a high volume of user-supplied annotations.

The CoVerity verifier [cov] has been successfully applied to large industrial as well as open source projects. While we lack a direct indication of the types of bugs found, an indirect indication is obtained by browsing through the fixes made in the Linux2.6 kernels due to bugs found by the CoVerity checker. Reports of these bugs are available online by browsing commit messages accompanying changes to the kernel source. The data suggest that almost all the overruns that were fixed involved statically allocated buffers and were all intraprocedural, in nature. On the other hand, we seek to find more complex bugs in smaller pieces of code, while relying on the user to selectively refine our initial assumptions on the calling environments of these modules by means of annotations.

The ESP analyzer of Das et al. uses path sensitive static analysis with heuristics for merging disjuncts [Das et al. 2002]. Recently, it has been used as an annotation checker to verify programmer written rich type annotations in the source code of large projects with remarkable success [Das 2006]. Our approach has the advantage of reporting concrete violations. Furthermore, our static analysis techniques systematically handle loops using abstract interpretation techniques.

Recently, numerous formalisms have focused on automatically generating models of the environment, based on automatic environment generation [Tkachuk et al. 2003], specification mining [Ammons et al. 2002; Kremenek et al. 2006], interface automata [de Alfaro and Henzinger 2001], and automata learning [Alur et al. 2005] and Datalog rules [Sankaranarayanan et al. 2008].

Model Checking. There have been past approaches that have attempted to model check programs for buffer overflows using various model checking techniques. The CBMC tool due to Clarke et al. [Clarke et al. 2004] uses SAT-based *bounded model checking* (BMC) to unroll a given program upto a fixed depth into a SAT problem, which is checked for the presence of a violation upto that depth [Biere et al. 1999]. We also use SAT-based BMC at its back-end. However, we use static analysis to vastly simplify the model and obtain invariants that vastly improve the performance of the model checker [Ganai and Gupta 2006].

Abstraction Refinement. Abstraction-refinement is commonly used in verification to achieve scalable verification of systems at the level of abstraction that suffices to establish the property or find a violation. A common paradigm for refining initially coarse abstractions based on concrete counterexamples is called *CEGAR* (Counterexample-Guided Abstraction Refinement) [Clarke et al. 2000; Ball et al. 2001; Kurshan 1994]. The *CEGER* approach is inspired, in part, by CEGAR. However, CEGAR approaches use spurious counterexamples to refine the abstract model of the program. In contrast, we use counterexamples to refine the calling environment of the function being verified, and also to model functions with missing sources. The model of the function is constructed up front and itself remains unchanged in each iteration.

The CEGER approach is not completely automated. Indeed, in a practical setting, the user (i) determines whether the counterexample is spurious, and (ii) provides the environment refinement. The model checker provides concrete traces and weakest preconditions to aid the user.

Predicate abstraction using automatic refinement has lead to important tools such as SLAM [Ball et al. 2001], BLAST [Henzinger et al. 2002], and many others. However, our own experience with predicate abstraction refinement (which is also implemented in our tool) suggests that for properties such as buffer overflows and strings, the automatic counterexample guided refinement leads to a large number of predicates. Therefore, we have been unable to apply predicate abstraction successfully for large programs to check buffer overflows. Furthermore, the model size increases exponentially at each iteration. The crucial distinction in our approach is that (a) our initial environment is not conservative and (b) we do not expect to finish with a suitable abstraction to prove correctness (which may require expensive analysis or too many refinements). We focus mainly on weeding out the spurious shallow bugs that are typically reported when the environment information is missing. We observe that “fixing” these witnesses eventually exposes the true bugs in the program.

8. CONCLUSION

We have presented a framework for checking buffer overflows in large C programs using accurate memory modeling, powerful static analyses, and efficient model checking. We have also proposed a user-driven environment refinement method, where automatically generated counterexamples and corresponding weakest preconditions provide support to the user for manually refining the environment constraints that are crucial for scalable verification of large programs. Our approach has shown real promise in its initial application by developers and found many

useful bugs in large software projects. We hope to expand the checker to handle higher-level properties specific to some application domains.

Acknowledgments

The authors gratefully acknowledge the expert guidance and the invaluable assistance provided by Mr. Kenjiroh Ikeda, Mr. Naoto Maeda, Mr. Yuusuke Hashimoto, Mr. Shinichi Iwasaki, and the NEC-SWED team, during the course of development of F-Soft. We gratefully acknowledge our team members Dr. Malay Ganai, Dr. Himanshu Jain, Dr. Vineet Kahlon, Dr. Chao Wang, Dr. Ziji Yang (James) Yang, Dr. Alex Zaks, Mr. Weihong Li and Ms. Nadia Papakonstantinou for their help in developing various parts of the F-SOFT tool chain.

REFERENCES

- Coverity inc. program verifier. <http://www.coverity.com>.
- PolySpace program analysis tool. <http://www.polyspace.com>.
- ALLAMIGEON, X., GODARD, W., AND HYMANS, C. 2006. Static Analysis of String Manipulations in Critical Embedded C Programs. In *SAS'06*. LNCS, vol. 4134. Springer, 35–51.
- ALUR, R., ČERNÝ, P., MADHUSUDAN, P., AND NAM, W. 2005. Synthesis of interface specifications for java classes. In *Proc. POPL*. ACM Press, 98–109.
- AMMONS, G., BODÍK, R., AND LARUS, J. R. 2002. Mining specifications. In *POPL*. 4–16.
- BALL, T., MAJUMDAR, R., MILLSTEIN, T., AND RAJAMANI, S. 2001. Automatic predicate abstraction of C programs. In *PLDI'01*. ACM Press, 203–213.
- BARNETT, M., FAHNDRICH, M., AND LOGOZZO, F. 2008. Foxtrot and Clousot: Language agnostic dynamic and static contract checking for .NET. Tech. Rep. MSR-TR-2008-105, Microsoft Research. August.
- BIERE, A., CIMATTI, A., CLARKE, E., AND ZHU, Y. 1999. Symbolic model checking without BDDs. In *Proc. TACAS*. Lecture Notes in Computer Science, vol. 1579. 193–207.
- BLANCHET, B., COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. 2003. A static analyzer for large safety-critical software. In *ACM SIGPLAN PLDI'03*. Vol. 548030. ACM Press, 196–207.
- BODIK, R., GUPTA, R., AND SARKAR, V. 2000. ABCD: Eliminating array bounds checks on demand. In *SIGPLAN PLDI'00*. ACM Press, 321–333.
- CHATTERJEE, S., LAHIRI, S. K., QADEER, S., AND RAKAMARIĆ, Z. 2007. A reachability predicate for analyzing low-level software. In *TACAS*. LNCS, vol. 4424/2007. Springer, 19–33.
- CHRISTENSEN, A. S., MOLLER, A., AND SCHWARTZBACH, M. I. 2003. Precise analysis of string expressions. In *SAS 2003*. LNCS, vol. 2694. Springer, 1–18.
- CLARKE, E., GRUMBERG, O., JHA, S., LU, Y., AND VEITH, H. 2000. Counterexample-guided abstraction refinement. In *Computer Aided Verification*. 154–169.
- CLARKE, E., KROENING, D., AND LERDA, F. 2004. A tool for checking ANSI-C programs. In *Proc. TACAS*. Lecture Notes in Computer Science, vol. 2988. Springer.
- COUSOT, P. AND COUSOT, R. 1976. Static determination of dynamic properties of programs. In *Proc. of the 2nd Intl. Symp. on Programming*. Dunod, France, 106–130.
- COUSOT, P. AND COUSOT, R. 1977. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Principles of Programming Languages*. 238–252.
- COUSOT, P. AND HALBWACHS, N. 1978. Automatic discovery of linear restraints among the variables of a program. In *ACM Principles of Programming Languages*. 84–97.
- COWAN, C., WAGLE, P., PU, C., BEATTIE, S., AND WALPOLE, J. 1999. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proc. DARPA Information Survivability Conference and Expo (DISCEX)*. IEEE.
- DAS, M. 2006. Unleashing the power of static analysis. In *SAS*. LNCS, vol. 4134. Springer, 1–2.

- DAS, M., LERNER, S., AND SEIGLE, M. 2002. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of Programming Language Design and Implementation (PLDI 2002)*. ACM Press, 57–68.
- DE ALFARO, L. AND HENZINGER, T. A. 2001. Interface automata. In *ESEC / SIGSOFT FSE*. ACM Press, 109–120.
- DOR, N., RODEH, M., AND SAGIV, M. 2003. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proc. PLDI*. ACM Press.
- GANAI, M. K. AND GUPTA, A. 2006. Accelerating high-level bounded model checking. In *ICCAD*. ACM, 794–801.
- GANAPATHY, V., JHA, S., CHANDLER, D., MELSKI, D., AND VITEK, D. 2003. Buffer overrun detection using linear programming and static analysis. In *Proc. ACM Conference on Computer Security (CCS'03)*. ACM.
- HENZINGER, T., JHALA, R., MAJUMDAR, R., AND SUTRE, G. 2002. Lazy abstraction. In *Symposium on Principles of Programming Languages*. 58–70.
- HIND, M. 2001. Pointer analysis: Haven't we solved this problem yet? In *PASTE*. ACM, 54–61.
- JHALA, R. AND MAJUMDAR, R. 2005. Path slicing. In *PLDI '05*. ACM, 38–47.
- J.R. BURCH, E.M. CLARKE, D.E. LONG, K.L. McMILLAN, AND D.L. DILL. 1994. Symbolic model checking for sequential circuit verification. *IEEE Trans. on CAD of Integrated Circuits and Systems* 13, 4, 401–424.
- KREMENEK, T., TWOHEY, P., BACK, G., NG, A. Y., AND ENGLER, D. R. 2006. From uncertainty to belief: Inferring the specification within. In *OSDI*. USENIX Association, 161–176.
- KURSHAN, R. 1994. *Computer-aided Verification of Coordinating Processes: the automata-theoretic approach*. Princeton University Press.
- LU, S., LI, Z., QIN, F., TAN, L., ZHOU, P., AND ZHOU, Y. 2005. BugBench: A benchmark for evaluating bug detection tools.
- MINÉ, A. 2001a. A new numerical abstract domain based on difference-bound matrices. In *PADO II*. LNCS, vol. 2053. Springer, 155–172.
- MINÉ, A. 2001b. The octagon abstract domain. In *AST 2001 in WCRE 2001*. IEEE. IEEE CS Press, 310–319.
- NATIONAL INSTITUTE FOR STANDARDS AND TECHNOLOGY. SAMATE reference dataset. Cf. <http://samate.nist.gov/SRD>, and http://samate.nist.gov/SRD/view_testcase.php?tID=1803.
- NECULA, G., McPEAK, S., AND WEIMER, W. 2002. CCured: Type-safe retrofitting of legacy code. In *Proc. Principles of Programming Languages (POPL'02)*. ACM.
- NISHIHARA, K., HATABU, A., AND MORIYOSHI, T. 2008. Parallelization of h.264 video decoder for embedded multicore processor. In *IEEE International Conference on Multimedia and Expo (ICME'08)*. 329–332.
- RUGINA, R. AND RINARD, M. 2000. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *PLDI*. ACM Press, 182–195.
- SANKARANARAYANAN, S., IVANČIĆ, F., AND GUPTA, A. 2007. Program analysis using symbolic ranges. In *SAS*. LNCS, vol. 4634. Springer, 366–383.
- SANKARANARAYANAN, S., IVANČIĆ, F., SHLYAKHTER, I., AND GUPTA, A. 2006. Static analysis in disjunctive numerical domains. In *SAS*. Lecture Notes in Computer Science, vol. 4134. Springer, 3–17.
- SANKARANARAYANAN, S., IVANČIĆ, F., AND GUPTA, A. 2008. Mining library specifications using inductive logic programming. In *ICSE*.
- SIMON, A. AND KING, A. 2002. Analyzing String Buffers in C. In *Proc. AMAST'02*. LNCS, vol. 2422. Springer, 365–379.
- TKACHUK, O., DWYER, M. B., AND PASAREANU, C. 2003. Automated environment generation for software model checking.
- TOKUOKA, H., MIYAZAKI, Y., AND HASHIMOTO, Y. 2007. C-language verification tool using formal methods “varvel”. *NEC Technical Journal: Special Issue on Embedded Software and Solutions* 2, 2, 34–37. cf. <http://www.nec.co.jp/techrep/en/journal/g07/n02/070209.html>.
- VENET, A. AND BRAT, G. P. 2004. Precise and efficient static array bound checking for large embedded c programs. In *PLDI*. ACM Press, 231–242.

- WAGNER, D., FOSTER, J., BREWER, E., , AND AIKEN, A. 2000. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. Network and Distributed Systems Security Conference*. ACM Press, 3–17.
- WANG, C., YANG, Z., GUPTA, A., AND IVANČIĆ, F. 2007. Using counterexamples for improving the precision of reachability computation with polyhedra. In *CAV*. Lecture Notes in Computer Science, vol. 4590. 352–365.
- WHALEY, J. AND LAM, M. S. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*. ACM, 131–144.
- YANG, Z., WANG, C., GUPTA, A., AND IVANČIĆ, F. 2006. Mixed symbolic representations for model checking software programs. In *MEMOCODE*. IEEE, 17–26.
- YAVUZ-KAHVECI, T., BARTZIS, C., AND BULTAN, T. 2005. Action language verifier, extended. In *CAV*. LNCS, vol. 3576. Springer, 413–417.
- ZITSER, M., LIPPMANN, R., AND LEEK, T. 2004. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proc. SIGSoft/FSE'04*. ACM, 97–106.