# Modular Checking for Buffer Overflows in the Large

Brian Hackett
Computer Science Department
Stanford University
Stanford, CA 94305
bhackett@stanford.edu

Manuvir Das, Daniel Wang, Zhe Yang
Center for Software Excellence
Microsoft Corporation
Redmond, WA 98052
{manuvir,daniwang,zhey}@microsoft.com

## Abstract

We describe an ongoing project, the deployment of a modular checker to statically find and prevent every buffer overflow in future versions of a Microsoft product. Lightweight annotations specify requirements for safely using each buffer, and functions are checked individually to ensure they obey these requirements and do not overflow. To date over 400,000 annotations have been added to specify buffer usage in the source code for this product, of which over 150,000 were automatically inferred, and over 3,000 potential buffer overflows have been found and fixed.

## 1. INTRODUCTION

We describe an ongoing project, the deployment of a modular checker to statically find and prevent every buffer overflow in future versions of a Microsoft product. Lightweight annotations specify requirements for safely using each buffer, and functions are checked individually to ensure they obey these requirements and do not overflow. To date over 400,000 annotations have been added to specify buffer usage in the source code for this product, of which over 150,000 were automatically inferred, and over 3,000 potential buffer overflows have been found and fixed.

Our checker is an integral part of the development process for this product. Developers receive immediate feedback about potential overflows in their code as part of the build process, and identified issues must be fixed before code can be added to mainline branches. Our checker is only one of many tools used to identify security issues. In combination with testing and runtime mitigation, these provide a robust defense against security exploits.

The key lesson from our experience has been the need to focus on *incremental* deployment of annotation-based checking. An all-or-nothing approach to deployment is incredibly risky, requiring huge initial effort for potentially minimum payoff. Incremental deployment allows teams to pay part of the cost of annotating to realize part of the benefit, which provides incentive for further annotation. Our system has been engineered from the ground up to provide incremental benefit for incremental cost. The principal components of our system are:

- SAL, an annotation language (Section 3). A language sufficiently general to specify any buffer interface will be needlessly complicated for simple interfaces. We *layer* the annotation language to reduce the startup effort needed to learn the language and understand each annotation. A base set of primitive annotations allow for general but verbose interface specifications. These primitives compose into high-level annotations, which compactly describe the most common interfaces. To annotate the vast majority of a code base, developers need only the high-level annotations, and learn to use the primitives only as necessary.

- SALInfer, an annotation inference engine (Section 4). Inference can dramatically reduce the effort needed to annotate legacy code. We opt for an aggressive and unsound inference, which can infer many correct annotations at the expense of inferring some false ones. Inference algorithms are fully customizable Datalog programs, allowing extensive code base specific tuning to maximize accuracy and coverage.

- ESPX, a modular overflow checker (Section 5). The checker must be able to find overflows accurately in any annotated, partially annotated, or unannotated code base. This can be accomplished by slicing results by confidence, presenting only the warnings most likely to correspond to overflows. This need for accuracy, and thus usability, means that false negatives, or missed overflows, are possible (our tool is used alongside additional checkers and runtime mitigation techniques, providing additional defense against missed overflows). As annotations are added, the level of confidence in warnings increases, and the number of false negatives decrease. Fully annotated code is comprehensively checked for overflows.

In Section 2 we begin with a brief example of the benefits and problems with modular checking. In Sections 3, 4, and 5 we describe the previous components in detail. In Section 6 we evaluate how well our system works in practice, and in Section 7 we discuss related work.

## 2. EXAMPLE

Consider how the program in Figure 1 is manually reviewed to be certain it is free from buffer overflows. Function `StringCopy` has an *implicit* interface for its parameter `dst`, requiring it to have a capacity of at least `size` elements. When reviewing `ProcessString`, we make sure that, when passed to `StringCopy` on line 10, `tmp` is at least `len` elements long. As long as `StringCopy` obeys its interface and writes at most `len` elements to `tmp`, `ProcessString` will not overflow.

`StringCopy` performs a character-by-character copy of `src` into `dst`, ensuring that the result is zero terminated. When

```
 1: void ProcessString(wchar_t *str)
 2: {
 3:     wchar_t buf[100];
 4:     wchar_t *tmp = &buf;
 5:
 6:     int len = wcslen(str) + 1;
 7:     if (len > 100)
 8:         Alloc(&tmp, len * sizeof(wchar_t));
 9:
10:     StringCopy(tmp, str, len);
11:     ...
12: }
13:
14: void StringCopy(wchar_t *dst, wchar_t *src, int size)
15: {
16:     wchar_t *dtmp = dst, *stmp = src;
17:
18:     for (int i = 0; i < size - 1 && *stmp; i++)
19:         *dtmp++ = *stmp++;
20:     *dtmp = 0;
21: }
22:
23: void Alloc(void **buf, int size)
24: {
25:     *buf = malloc(size);
26: }
```

**Figure 1: Example application**

reviewing `StringCopy`, we make sure that at most `size` elements are written to `dst`. As long as the interface is obeyed by its callers, `StringCopy` will not overflow.

Now, suppose the allocation size on line 8 is changed to simply `len`. Then `ProcessString` may call `StringCopy` with a buffer only `len / sizeof(wchar_t)` elements long, and since `ProcessString` does not obey the interface on `StringCopy`, `StringCopy` may overflow. Alternatively, suppose the condition on line 18 is changed to simply `*stmp`. Then `StringCopy` may write more than `len` elements to `dst`, and since `StringCopy` does not obey its own interface, `ProcessString` may overflow.

Even very thorough manual review will miss many overflows such as these, and static checking is necessary for comprehensive safety guarantees. Effectively performing this checking requires knowledge of the implicit interface between `ProcessString` and `StringCopy`, and an understanding of how each function obeys that interface. A checker local to each modified function will not learn the interface, while a checker global to the entire code base will not understand the functions well enough without sacrificing scalability.

Modular checking offers a solution. We use annotations to make the interface on `StringCopy` explicit in the code (shown in Figure 3). The modified code is locally checked against these interfaces, and if it passes we guarantee it introduces no new overflows. A modular checker can precisely and comprehensively check code bases for overflows. The drawback is the need for annotations. Code will only be annotated if there is demonstrable benefit to doing so. As such, we provide checking benefit for unannotated and partially annotated code bases, and ease the annotation process through effective inference and a clean, usable annotation language.

## 3. ANNOTATION LANGUAGE

We use annotations to describe requirements on buffer pointers (in C, all pointers reference buffers and may overflow; all pointers must then be annotated). For each buffer pointer passed into or returned by each function, what is the function's *interface* with the buffer, i.e. what are the assumptions and guarantees the function makes about the buffer's length and contents?

| Where | Level | Property |
|---|---|---|
| `pre` | $\epsilon$ | `notnull` |
| `post` | `deref` | `eread(`$e_l$`)` |
| | | `bread(`$e_l$`)` |
| | | `zread` |
| | | `ewrite(`$e_s$`)` |
| | | `bwrite(`$e_s$`)` |
| | | `zwrite` |

**Table 1: Primitive annotation table**

| Usage | Optional | Initialized | Capacity |
|---|---|---|---|
| `in` | $\epsilon$ | $\epsilon$ | $\epsilon$ |
| `out` | `opt` | `ecount(`$e_l$`)` | `ecap(`$e_s$`)` |
| `inout` | | `bcount(`$e_l$`)` | `bcap(`$e_s$`)` |
| `ret` | | `zterm` | |
| `dret` | | | |

**Table 2: Buffer annotation table**

Design of the annotation language has two chief goals, *expressiveness* and *conciseness.*

- Expressive annotations accurately describe a wide variety of buffer interfaces. If a buffer interface cannot be accurately annotated, checking the correctness of that function and its callers requires the interface to be changed. This is often impossible or impractical due to the need for backward compatibility, and even if possible, presents a huge cost to developers.

- Concise annotations succinctly describe buffer interfaces. Verbose annotations have several costs: individual annotations will be difficult to understand and difficult to write, and, most importantly, easy to get wrong. Investment in annotating a code base is largely wasted if the annotations are frequently incorrect and inconsistent.

Expressiveness and conciseness are, unfortunately, often at odds with each other. We achieve a combination of expressiveness and conciseness by *layering* the annotation language. An expressive layer of *primitive annotations* describe primitive properties held by particular buffers at function entry or exit. Several primitive annotations are necessary to completely describe most interfaces. A second, concise layer of *buffer annotations* describe all common interfaces. Only one buffer annotation is required for each interface, simplifying and reducing errors in the annotation process. In practice, buffer annotations are used to describe all applicable interfaces. For more complicated cases, primitive annotations are effective.

### 3.1 Primitive Annotations

Each primitive annotation describes a particular property that must be held by particular buffer pointers at either entry to or exit from the function. These properties form a natural extension of the pointer's type, and thus we represent primitive annotations as qualifiers on the types of pointer parameters and return values. Types are an extremely succinct way of specifying interfaces, and one with which developers are already familiar. We present the full grammar for primitive annotations here. Example annotations are shown in Figure 2, and discussed in Example 1.

The primitive properties we are interested in are `NULL`-ness and the readable and writable extents of each pointer. Memory is initially writable, and made readable by initializing it. Readable and writable extents are permissions: they may not describe the entire buffer (for example, only part of a buffer may be marked as writable), and they may not describe all permissions which are held (for example, a function may only have read permission for a buffer, even if its caller also has write permission).

The possible primitive annotations are given by Table 1. Each column in the table identifies one aspect of a primitive annotation, and any primitive annotation may be formed by choosing one value from each column ($\epsilon$ indicates an empty value) and combining them together using '_' as a separator. For example, choosing `pre` and `eread(`$e_l$`)` yields '`__pre_eread(`$e_l$`)`'. The meaning of each column and value is as follows:

- Where: Which point the property holds at.
  - `pre`: At function entry.
  - `post`: At function exit.
- Level: Which pointer $v$ the property holds for, relative to the annotated parameter/returned pointer $p$.
  - $\epsilon$: The property holds for $p$.
  - `deref`: The property holds for all pointers at $*p$.
- Property: Which property holds for the pointer $v$.
  - `notnull`: $v$ is not `NULL`.
  - `eread(`$e_l$`)`: $v$ is `NULL` or readable to $e_l$ elements. Expressions $e$ are arithmetic expressions over constants, parameters, and parameter dereferences.
  - `bread(`$e_l$`)`: $v$ is `NULL` or readable to $e_l$ bytes.
  - `zread`: $v$ is `NULL` or readable to a zero terminator.
  - `ewrite(`$e_s$`)`: $v$ is `NULL` or writable to $e_s$ elements.
  - `bwrite(`$e_s$`)`: $v$ is `NULL` or writable to $e_s$ bytes.
  - `zwrite`: $v$ is `NULL` or writable to a zero terminator.

As each primitive annotation is a well-defined precondition or postcondition on the function, any combination of primitive annotations on a pointer leads to a checkable interface.

Example 1. Consider again the application in Figure 1. Figure 2 shows the primitive annotations for each pointer used by each function.

`ProcessString` takes a string `str`, copies it into a scratch buffer and does not write to it. The string must be non-`NULL` and readable (but not writable) up to a zero terminator. The correct annotations are `__pre_notnull` and `__pre_zread`.

`StringCopy` takes a string `src` and copies as much of it into `dst` as possible, not exceeding `size` characters and zero terminating `dst`. `src` is annotated as `__pre_notnull` and `__pre_zread` as before. `dst` must be a non-`NULL` buffer with `size` elements, writable but not readable at entry, and must be initialized with a zero terminated string at exit. The correct annotations are `__pre_notnull`, `__pre_ewrite(size)`, and `__post_zread`.

`Alloc` returns an uninitialized buffer with `size` bytes by assigning to `*buf`. Both `buf` and `*buf` must be annotated. `buf` must be non-`NULL` and will filled in with a single value. The correct annotations are `__pre_notnull`, `__pre_ewrite(1)`, and `__post_eread(1)`. `*buf` will be filled in with a non-`NULL` buffer with `size` uninitialized bytes. The correct annotations are `__post_deref_notnull` and `__post_deref_bwrite(size)`. □

```
void ProcessString(
  __pre_notnull __pre_zread wchar_t *str);

void StringCopy(
  __pre_notnull __pre_ewrite(size) __post_zread wchar_t *dst,
  __pre_notnull __pre_zread wchar_t *src,
  int size);

void Alloc(
  __pre_notnull __pre_ewrite(1) __post_eread(1)
  __post_deref_notnull __post_deref_bwrite(size) void **buf,
  int size);
```

**Figure 2: Primitive annotations for Figure 1**

```
void ProcessString(__in_zterm wchar_t *str);

void StringCopy(__out_zterm_ecap(size) wchar_t *dst,
                __in_zterm wchar_t *src,
                int size);

void Alloc(__out __dret_bcount_bcap(0,size) void **buf,
           int size);
```

**Figure 3: Buffer annotations for Figure 1**

The chief drawback of using primitive annotations is the large number that may be required for simple interfaces. In Figure 2, for example, three annotations are required simply to specify that parameter `buf` of `Alloc` is filled in with one value. An effective annotation system must describe such simple interfaces clearly and concisely. To accomplish this, we use buffer annotations.

## 3.2 Buffer Annotations

Each buffer annotation fully describes a buffer interface, and includes all properties needed to safely use a buffer pointer. As with primitive annotations, buffer annotations are represented as type qualifiers. We present the full grammar for buffer annotations here. Example annotations are shown in Figure 3, and discussed in Example 2.

The main concept which allows for concise interface description is *usage*. Buffer pointer parameters are naturally either `in`, `out`, or `inout`, according to whether they move data into or out of the function, and this abstraction is somewhat clumsy to encode with preconditions and postconditions.

The possible buffer annotations are given by Table 2, and, as with primitive annotations, individual buffer annotations are formed by choosing one value from each column and combining them. The meaning of each column and value is as follows:

- Usage: How the buffer is used by the function.
  - `in`: Passed in and read from.
  - `out`: Passed in and written to.
  - `inout`: Passed in and read/written to.
  - `ret`: Returned via return value.
  - `dret`: Returned via an `out` parameter.
- Optional: Whether the buffer pointer may be `NULL`.
  - $\epsilon$: Must be non-`NULL`.

  - `opt`: May be `NULL`.

- Initialized: How much of the buffer is initialized. This is a lower bound on the initialized amount (more could be initialized, but not less). The initialized length may change during execution for `out` and `inout` buffers. For `out` parameters, only the initialized length at exit is described. For `inout` parameters, the initialized length at both entry and exit is described.
  - $\epsilon$: One element.
  - `ecount(`$e_l$`)`: $e_l$ elements.
  - `bcount(`$e_l$`)`: $e_l$ bytes.
  - `zterm`: Up to the first zero terminator.

- Capacity: The total allocated capacity of the buffer. This is a lower bound on the capacity.
  - $\epsilon$: The initialized amount (at function entry for `inout` buffers).
  - `ecap(`$e_s$`)`: $e_s$ elements.
  - `bcap(`$e_s$`)`: $e_s$ bytes.

A total of 120 buffer annotations may be formed from Table 2, which collectively describe the simplest and most common buffer interfaces in use.

Example 2. Consider again the application in Figure 1. For each pointer parameter and return value in this application, we must pick appropriate values from each column of Table 2, combining them to form the correct annotation.

`ProcessString` takes a string `str`, copies it into a scratch buffer and does not write to it. The string is then `in`, and since it must be zero terminated, is also `zterm`. The correct annotation is `__in_zterm`.

`StringCopy` takes a string `src` and copies as much of it into `dst` as possible, not exceeding `size` characters and zero terminating `dst`. `src` is `__in_zterm` as before. `dst` is `out`, `zterm` when initialized at exit, and `ecap(size)` since up to `size` characters may be written. The correct annotation is `__out_zterm_ecap(size)`.

`Alloc` returns an uninitialized buffer with `size` bytes by assigning to `*buf`. `buf` is simply `__out`. `*buf` is returned through `buf`, and thus `dret`, has `bcount(0)` bytes initialized but `bcap(size)` bytes allocated, so the correct annotation is `__dret_bcount_bcap(0,size)`. □

Each buffer annotations is defined as a set of primitive annotations (in our implementation, each buffer annotation is a C macro expanding to the corresponding primitives). For example, `__out` is equivalent to `__pre_notnull`, `__pre_ewrite(1)`, and `__post_eread(1)`. This provides a clear definition of each buffer annotation and narrows the set of annotations the checker must understand to simple preconditions and postconditions.

A few buffer annotations cannot be represented with primitive annotations, and cannot be checked for correctness. In particular, `__out_zterm` annotates a buffer for which the function is given carte blanche to write a string of any size. We disallow these unsafe buffer annotations; calls to functions which require them (well-known examples include the C standard library functions `strcpy`, `strcat`, `gets`, and `sprintf`) are being removed through a separate process not described here, and are ignored by our checker. Many overflows occur due to functions with unsafe interfaces; for example, 5 of the 14 overflows studied in [18] are due to incorrect usage of `strcpy` and `sprintf`.

## 4. ANNOTATION INFERENCE

Comprehensively checking legacy code for overflows requires that code to be annotated. For large code bases, the cost of legacy annotation can be very high. Annotation inference offers a way to avoid much of this cost. Rather than requiring manual annotation of significant portions of the code base before it can be checked, we annotate as much as possible automatically, and check the code for overflows and incorrect annotations.

The chief requirement for an inference tool is scalability. Legacy code bases may contain tens of millions lines of code, and inference tools are useless for code bases they cannot generate results for. Beyond scalability, inference effectiveness is measured by its *accuracy* and *coverage*.

- Accuracy is the fraction of inferred annotations that are correct. Incorrect annotations are caught by the checker, but the error messages can be confusing and the developer cost of fixing an incorrect annotation is often higher than that of adding the correct annotation initially.

- Coverage is the fraction of correct annotations that are inferred. Incomplete annotations have minimal impact on checker effectiveness, and require significantly more developer effort to fully annotate the code base.

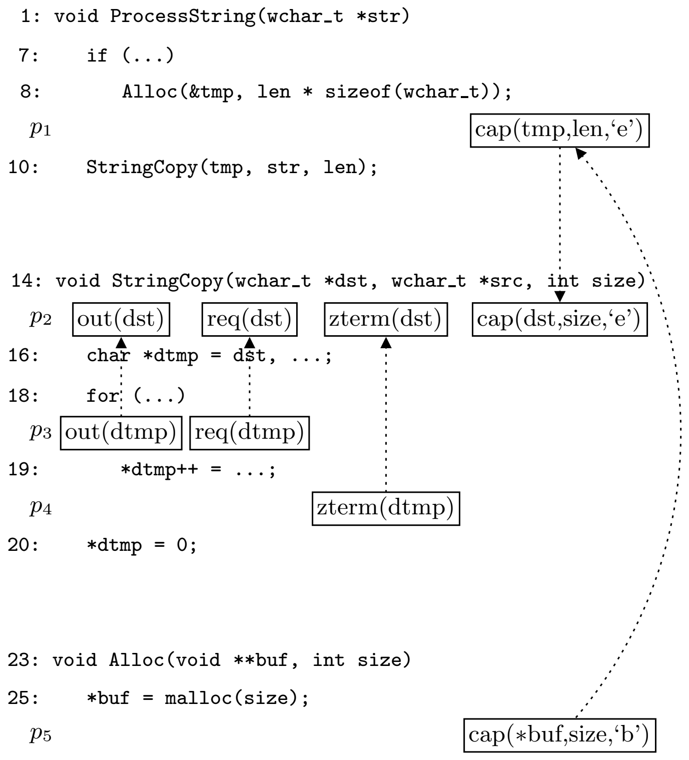
Under the fixed requirement for scalability, accuracy and coverage are at odds with each other. More accurate techniques infer less information, compromising coverage, and higher coverage techniques infer more information, compromising accuracy. Both can be improved through more precise analysis, but scalability is then impacted. To optimize these trade/offs, we use a fully customizable inference engine, which can be rapidly tuned to generate effective results for particular code bases and kinds of annotations.

To infer buffer annotations, we infer each aspect (usage, `NULL`-ness, initialized length and capacity) independently, aggressively and unsoundly propagating *states*, particular properties holding of particular values at particular program points, through the code base. Propagation rules are fully customizable, controlled by identifying source code *patterns* using rules which are themselves fully customizable. Constraining or removing rules decreases the amount of inferred information, increasing accuracy, while expanding or adding rules increases the amount of inferred information, increasing coverage.

Example 3. Consider again the application in Figure 1. Parameter `dst` for `StringCopy` needs the buffer annotation `__out_zterm_ecap(size)`. This encapsulates four pieces of information: `dst` is only written to, must be non-`NULL`, will be zero terminated, and must be at least `size` elements long. Figure 4 shows how each of these properties may be separately inferred and combined into the correct annotation.

Each out state indicates a value that may be written to later in the function. `StringCopy` writes to `dtmp` on line 19 (and 20), so out(`dtmp`) is added beforehand at program point $p_3$. We propagate this backwards through the assignment on line 16 from `dst` to `dtmp`, adding out(`dst`) at the entry point $p_2$ of `StringCopy`. From this, we infer that `dst` is an `out` parameter. (If `dst` was also read from, we would also infer in(`dst`) at $p_2$ to yield `inout`).

Each req state indicates a value that is required to be non-`NULL`; it may be accessed later in the function without being



```
1: void ProcessString(wchar_t *str)
7:    if (...)
8:       Alloc(&tmp, len * sizeof(wchar_t));
10:   StringCopy(tmp, str, len);

14: void StringCopy(wchar_t *dst, wchar_t *src, int size)
16:   char *dtmp = dst, ...;
18:   for (...)
19:      *dtmp++ = ...;
20:   *dtmp = 0;

23: void Alloc(void **buf, int size)
25:   *buf = malloc(size);
```

**Figure 4: Inferred properties for Figure 1**

checked against `NULL` first. `dtmp` is accessed on line 19, so req(`dtmp`) is added at $p_3$. Since there is no prior guard against `NULL`, we propagate back to $p_2$ and infer that `StringCopy` requires `dst` to be non-`NULL`.

Each zterm state indicates a value that may be zero terminated later in the function. `StringCopy` writes a zero to `dtmp` on line 20. As `dtmp` is also a string type, we add zterm(`dtmp`) at $p_4$, propagate this back to $p_2$ and infer that `StringCopy` should zero terminate `dst`.

Each cap state indicates buffer-size pairs of values, with a size kind of either ‘e’ (elements) or ‘b’ (bytes). On line 25, `Alloc` copies into `buf` a malloc'ed buffer with `size` bytes. cap(*`buf`,`size`,‘b’) is added at $p_5$, and propagated through the call on line 8 to cap(`tmp`,`len`,‘e’) at $p_1$, changed to an element count by the multiplication `len * sizeof(wchar_t)`. We propagate through the call on line 10 to cap(`dst`,`size`,‘e’) at $p_2$, and infer that `StringCopy` expects `dst` to be `size` elements long. □

Note in Example 3 that the propagation method for buffer-size relationships is unsound; at the call on line 10 it is known that `tmp` and `len` are a buffer-size pair, but there is no guarantee at entry to `StringCopy` that this always holds. Observing the lengths of buffers actually passed to a function is a simple and reliable method for guessing the required length, and such leaps of logic are crucial for effective inference. However, this process may break down on certain code base specific patterns. For example, some API's take either Unicode or ASCII strings, depending on the value of some flag, and the resulting path-sensitive behavior must be recognized and avoided.

We accomplish this through customization, tuning the inference to behave effectively for the targeted code base. Each inference algorithm is simply a Datalog program, and the inference engine is a custom Datalog solver, built under the requirements of performing interprocedural analysis over millions of lines of code.

Each inference algorithm uses two input predicates prop and edge to specify, respectively, all raw syntax and control flow information for the targeted code base. The algorithm rules then iteratively build up high level semantic information from this: pattern rules derive semantic ‘facts’ about particular syntax trees, state rules derive and propagate facts about the program state at particular program points, and output rules derive annotations, facts about the code base itself.

Example 4. An inference algorithm for buffer/size relationships is given in Figure 5. Buffer information is seeded at `malloc` call sites and propagated forward and backward through assignments, using the feasible relationships at function entry points to infer capacity annotations.

Rules 1-7 identify syntax tree patterns for buffer propagation. Patterns are constructed using the prop predicate, which identifies the value of a named property of a syntax tree $t$. A fixed set of property names allow inference algorithms to extract all available information. For example, rule 1 identifies `malloc` call sites which return buffer $b$ with size $s$. Property ‘kind’ picks out the base kind of the tree (‘call’), while ‘callee’ identifies the callee, ‘retval’ identifies the return value, and ‘arg0’ identifies the first argument.

Rules 8-16 propagate cap states through the target code base. States are constructed using (optionally) other states, patterns and the edge predicate, which identifies forward control flow edges between program points $p_0$ and $p_1$, picking out the syntax tree $t$ for the action taken over that edge. Rule 8 seeds cap by deriving states after every malloc site. Rules 9-12 propagate cap forward across copies of the buffer or size, and backward across assignments into the buffer or size, while rules 13-14 propagate forward and backward across assignments that scale the size between an element or byte count. Finally, rules 15-16 propagate cap across all edges that do not kill (assign into) either the buffer or size.

Rule 17 infers fcap output annotations, which pick out buffer/size parameter pairs taken by functions in the code base. These are simply those cap states feasible at entry to the functions. □

## 5. MODULAR CHECKER

Given the buffer interfaces in a code base, checking the safety of each function $f$ with callees $g$ is, in the end, fairly straightforward: assuming $f$'s preconditions and each $g$'s postconditions hold, is each access safe and are $f$'s postconditions and each $g$'s preconditions guaranteed to hold? The chief difficulty arrives when, due to incomplete information, we cannot be sure that the code is definitely safe, nor that it is definitely unsafe.

The checker's goal is to miss no overflows in passed code. For a property as pervasive and complex as memory usage, this goal is not immediately realizable on an existing code base. To yield useful results on unannotated or partially annotated code bases, the checker must be confident that reports are genuinely errors. We briefly describe the checker algorithm here. In Section 5.1 we describe how confidence tunes the checker to the degree of annotation.

For each function, the checker performs an exhaustive

**Predicates**

| | | |
|---|---|---|
| input | prop($t$,$x$,$v$) | Property $x$ of syntax tree $t$ is value $v$. |
| input | edge($p_0$,$p_1$,$t$) | Program points $p_0$ and $p_1$ are connected by a control flow edge with syntax tree $t$. |
| pattern | malloc($t$,$b$,$s$) | $t$ is a call to malloc returning buffer $b$ with size $s$. |
| pattern | assign($t$,$l$,$r$) | $t$ is an assignment from $r$ to $l$. |
| pattern | scale($t$,$v$) | $t$ is a multiplication of $v$ by some value. |
| pattern | sassign($t$,$l$,$r$) | $t$ is an assignment from a scaled $r$ to $l$. |
| pattern | nokill($t$,$b$,$s$) | $t$ is a syntax tree that does not erase the value of $b$ or $s$. |
| pattern | entry($t$,$f$) | $t$ is the syntax tree for the entry edge of function $f$. Functions have unique entry and exit flow edges. |
| state | cap($p$,$b$,$s$,$k$) | At program point $p$, buffer $b$ has size $s$ with kind $k$. Kinds are either ‘e’ (elements) or ‘b’ (bytes). |
| output | fcap($f$,$b$,$s$,$k$) | Function $f$ takes buffer/size parameters $b$ and $s$ with kind $k$. |

**Pattern Rules**

1. malloc($t$,$b$,$s$) :- prop($t$,‘kind’,‘call’), prop($t$,‘callee’,‘malloc’), prop($t$,‘retval’,$b$), prop($t$,‘arg0’,$s$).

2. assign($t$,$l$,$r$) :- prop($t$,‘kind’,‘assign’), prop($t$,‘lhs’,$l$), prop($t$,‘rhs’,$r$).

3. scale($t$,$v$) :- prop($t$,‘kind’,‘times’), prop($t$,‘lhs’,$v$).
4. scale($t$,$v$) :- prop($t$,‘kind’,‘times’), prop($t$,‘rhs’,$v$).
5. sassign($t$,$l$,$r$) :- assign($t$,$l$,$t'$), scale($t'$,$r$).

6. nokill($t$,$b$,$s$) :- ∼assign($t$,$b$,_), ∼assign($t$,$s$,_).

7. entry($t$,$f$) :- prop($t$,‘kind’,‘entry’), prop($t$,‘func’,$f$).

**State Rules**

8. cap($p$,$b$,$s$,‘b’) :- edge(_,$p$,$t$), malloc($t$,$b$,$s$).

9. cap($p_1$,$l$,$s$,$k$) :- cap($p_0$,$r$,$s$,$k$), edge($p_0$,$p_1$,$t$), assign($t$,$l$,$r$).
10. cap($p_1$,$b$,$l$,$k$) :- cap($p_0$,$b$,$r$,$k$), edge($p_0$,$p_1$,$t$), assign($t$,$l$,$r$).
11. cap($p_0$,$r$,$s$,$k$) :- cap($p_1$,$l$,$s$,$k$), edge($p_0$,$p_1$,$t$), assign($t$,$l$,$r$).
12. cap($p_0$,$b$,$r$,$k$) :- cap($p_1$,$b$,$l$,$k$), edge($p_0$,$p_1$,$t$), assign($t$,$l$,$r$).

13. cap($p_1$,$b$,$l$,‘b’) :- cap($p_0$,$b$,$r$,‘e’), edge($p_0$,$p_1$,$t$), sassign($t$,$l$,$r$).
14. cap($p_0$,$b$,$r$,‘e’) :- cap($p_1$,$b$,$l$,‘b’), edge($p_0$,$p_1$,$t$), sassign($t$,$l$,$r$).

15. cap($p_1$,$b$,$s$,$k$) :- cap($p_0$,$b$,$s$,$k$), edge($p_0$,$p_1$,$t$), nokill($t$,$b$,$s$).
16. cap($p_0$,$b$,$s$,$k$) :- cap($p_1$,$b$,$s$,$k$), edge($p_0$,$p_1$,$t$), nokill($t$,$b$,$s$).

**Output Rules**

17. fcap($f$,$b$,$s$,$k$) :- cap($p$,$b$,$s$,k), edge(_,$p$,$t$), entry($t$,$f$).

**Figure 5: Datalog program for inferring buffer/size relationships**

path exploration based on the RHS algorithm [12]. At each program point, we compute the feasible sets of linear constraints over constant values, locations $l$ (variables and heap values), and pointer capacities (denoted as bcap($l$)). If at a control flow join point multiple sets of linear constraints are feasible, they are not merged but are separately analyzed. This reduces information loss, simplifies constraint representation, and, most importantly, allows us to easily generate traces for developer inspection. To ensure termination, the set of constraints is widened (made more general) according to heuristics along loop back edges.

At function entry, constraints are generated and assumed from the preconditions. For example, if `__pre_bwrite(len)` holds for parameter `buf`, the constraint bcap(buf) $\geq$ len is generated. At each dereference on a pointer location $l$, we generate constraints describing whether the accessed portion is in bounds, and check whether they always hold. For example, if buf[2] is accessed and elements of buf are 2 bytes wide, the constraint bcap(buf) $\geq 6$ is checked. If this constraint is not guaranteed to hold, an overflow is possible. At each function call, constraints are generated from the callee preconditions and checked, while further constraints are generated and assumed from the callee postconditions. Finally, at function exit constraints are generated from the postconditions and checked.

Example 5. Consider again the application from Figure 1. Figure 6 shows the constraints generated for a modified version of function `ProcessString`: the allocation size on

**ProcessString path 1, string length $\leq$ 100**

`3: wchar_t buf[100];`
`4: wchar_t *tmp = &buf;`
$$\text{bcap}(\text{tmp}) = 200$$
`6: int len = wcslen(str) + 1;`
$$\text{bcap}(\text{tmp}) = 200$$
`7: assume(len <= 100);`
$$\text{bcap}(\text{tmp}) = 200, \text{len} \leq 100$$
`10: StringCopy(tmp, str, len);`
$$\textbf{assert}(\text{bcap}(\text{tmp}) \geq \text{len} * 2)$$
$$\Box\ \text{bcap}(\text{tmp}) = 200 = 100 * 2 \geq \text{len} * 2 \quad \textbf{pass}$$

**ProcessString path 2, string length > 100**

`3: wchar_t buf[100];`
`4: wchar_t *tmp = &buf;`
$$\text{bcap}(\text{tmp}) = 200$$
`6: int len = wcslen(str) + 1;`
$$\text{bcap}(\text{tmp}) = 200$$
`7: assume(len > 100);`
$$\text{bcap}(\text{tmp}) = 200, \text{len} > 100$$
`8: Alloc(&tmp, len` ~~`* sizeof(wchar_t)`~~`);`
$$\text{bcap}(\text{tmp}) \geq \text{len}, \text{len} > 100$$
`10: StringCopy(tmp, str, len);`
$$\textbf{assert}(\text{bcap}(\text{tmp}) \geq \text{len} * 2)$$
$$\Diamond\ \text{bcap}(\text{tmp}) = \text{len} < \text{len} * 2 \quad \textbf{fail}$$

**Figure 6: Checked paths from Figure 1**

line 8 has been changed to `len`, introducing an overflow.

There are two paths through `ProcessString`, depending on whether buffer `tmp` is stack or heap allocated at the call to `StringCopy` on line 10. On path 1, `tmp` is stack allocated. Setting `tmp` to `&buf` on line 4 generates the constraint $\mathsf{bcap}(\mathrm{tmp}) = 200$, as `buf` is a 200 byte buffer. Taking the false branch of the if statement on line 7 adds the constraint $\mathrm{len} \leq 100$. At the call to `StringCopy` on line 10, we must ensure that `__pre_ewrite(size)` holds for parameter `dst`, and thus for the arguments that $\mathsf{bcap}(\mathrm{tmp}) \geq \mathrm{len} * 2$. This constraint is implied by $\mathsf{bcap}(\mathrm{tmp}) = 200$ and $\mathrm{len} \leq 100$, so `StringCopy`'s precondition is satisfied and the call is safe.

On path 2, `tmp` is heap allocated, with half the required size. At the call to `Alloc` on line 8, we must ensure that `__pre_ewrite(1)` holds for parameter `buf`, and thus for the arguments that $\mathsf{bcap}(\&\mathrm{tmp}) \geq 4$, which must hold since tmp is a stack variable. We then assume `__post_deref_bwrite(size)` for `buf`, adding the argument constraint $\mathsf{bcap}(\mathrm{tmp}) \geq \mathrm{len}$ and removing $\mathsf{bcap}(\mathrm{tmp}) = 200$. At the call to `StringCopy` on line 10, we must again ensure that $\mathsf{bcap}(\mathrm{tmp}) \geq \mathrm{len} * 2$. If $\mathsf{bcap}(\mathrm{tmp}) = \mathrm{len}$ and $\mathrm{len} > 0$ this does not hold, and since the precondition may not be satisfied, the call is not safe and a warning is issued. □

## 5.1 Checker Confidence

To prove an access safe, the checker requires knowledge about the access external to the function and an understanding of the function's behavior with respect to the access. With complete knowledge and sufficient understanding by the checker, all accesses which cannot be proved safe are likely bugs and reported as such. Where the checker has an incomplete or imprecise model of the code, however, the checker reports only the accesses it is most confident of being unsafe. This opens a path to soundness: the value provided by identifying bugs gives incentive to improve knowledge and understanding, gradually eliminating false negatives.

External knowledge about accesses is conveyed through the buffer interfaces annotated for each function. Deficiencies here are categorized as follows:

- Unannotated interfaces. Absent any knowledge of the read and write extents for a pointer, no meaningful constraints can be generated and the checker can say nothing other than that it needs an annotation.
- Unannotatable interfaces. Some interfaces cannot be completely or correctly described by primitive annotations. Examples include unsafe string functions and functions which rely on particular conditions to hold or particular properties to hold only conditionally, as well as interfaces on structure field pointers or global pointers.

We are currently working to support conditional annotations and annotations on structure fields and global variables, greatly reducing the number of unannotatable interfaces. Beyond this, the key challenge is to actually get pointers annotated. Since proving an access safe often requires knowledge both of the function's interfaces and its callee's interfaces, we find that incremental annotation is best done in a targeted fashion. Particular classes of interfaces, such as those in new code, in particular parts of the code base, or on particular kinds of buffers (tainted buffers, string buffers, etc.), are in turn exhaustively annotated and checked.

Even if a particular buffer interface is annotated, the checker's confidence that accesses on it are unsafe is determined by how well the constraints on that access are understood by the checker. Sources of analysis imprecision are categorized as follows:

- Complex operations. Many program operations cannot be precisely represented as constraints. For example, after the statement `x = y & z;` we know only that $x < y$ and $x < z$. Code can be proved safe even if it is not modelled precisely, but where the imprecision is relevant to safety the checker will have more false positives and degrades its confidence accordingly.
- Loop widening. Many loops can run for an unbounded number of iterations, and to ensure checker termination constraints are widened at loop back edges. If the loop invariants relevant to safety are retained during widening, the code can be proved safe, but if not then false positives increase and confidence degrades.

Accesses whose constraints cannot be modelled precisely have lower confidence of being unsafe, and yet may be bugs. To reason about checker imprecision, we recover dependency information from use/mod sets for complex operations and loop conditions. Checker warnings are then prioritized based on the amount and nature of the imprecision involved in constraining the access, allowing for either minimal false positives with high priority warnings, or minimal false negatives (excepting unannotated pointers) with all warnings. The different warning levels we use are as follows, in decreasing order of priority.

1. No information constraining the access is lost. If the path containing the access is feasible, the access is provably unsafe and may overflow.
2. Some information constraining the access is lost, but appears unrelated to the pointer's length. If all lost information about the access is unrelated to the length, and the remainder cannot be used to prove the access safe, the access is a likely overflow.
3. Some information relating the access with the pointer length is lost. The access cannot be proved safe, but little evidence exists that it is unsafe either.

Checker precision is improved in a targeted fashion according to the distribution of bugs and false positives. Particular patterns of bugs are identified and handled more precisely in the checker to boost their priority, whereas particular classes of false positives are identified and handled more precisely to prove them safe and increase the overall quality of the warnings at that level.

## 6. RESULTS

We briefly sketch the current deployment status of the annotation language, inference, and checker over the Microsoft product. In Sections 6.1 to 6.3 we gauge the effectiveness of each technology.

Annotations are added through two main mechanisms. First, new code must be fully annotated before being checked in. Using concise buffer annotations, this burden is comparable to that of adding types. Second, annotations are used

| Count | Buffer Annotation | C | Primitive Annotations | C |
|---|---|---|---|---|
| 240311 | __in | 1 | __pre_notnull __pre_eread(1) | 4 |
| 53407 | __out | 1 | __pre_notnull __pre_ewrite(1) __post_eread(1) | 6 |
| 29807 | __in_opt | 2 | __pre_eread(1) | 2 |
| 28735 | __inout | 1 | __pre_notnull __pre_ewrite(1) __pre_eread(1) | 6 |
| 13567 | __out_ecap($e_s$) | 2 | __pre_notnull __pre_ewrite($e_s$) __post_eread(1) | 6 |
| 12907 | __in_ecount($e_l$) | 2 | __pre_notnull __pre_eread($e_l$) | 4 |
| 7560 | __out_opt | 2 | __pre_ewrite(1) __post_eread(1) | 4 |
| 6708 | __in_bcount($e_s$) | 2 | __pre_notnull __pre_bread($e_l$) | 4 |
| 4227 | __out_bcap($e_l$) | 2 | __pre_notnull __pre_bwrite($e_s$) __post_eread(1) | 6 |
| 2913 | __inout_opt | 2 | __pre_ewrite(1) __pre_eread(1) | 4 |
| 2756 | __out_opt_ecap($e_s$) | 3 | __pre_ewrite($e_s$) __post_eread(1) | 4 |
| 2651 | __inout_ecap($e_s$) | 2 | __pre_notnull __pre_ewrite($e_s$) __pre_eread(1) | 6 |
| 1589 | __in_opt_ecount($e_l$) | 3 | __pre_eread($e_l$) | 2 |
| 1302 | __out_opt_bcap($e_s$) | 3 | __pre_bwrite($e_s$)__post_eread(1) | 4 |
| 998 | __in_opt_bcount($e_l$) | 3 | __pre_bread($e_l$) | 2 |
| 739 | __out_ecount_ecap($e_l$,$e_s$) | 3 | __pre_notnull __pre_ewrite($e_s$) __post_eread($e_l$) | 6 |
| 682 | __inout_bcap($e_s$) | 2 | __pre_notnull __pre_bwrite($e_s$) __pre_eread(1) | 6 |
| 581 | __out_opt_bcount_bcap($e_l$,$e_s$) | 4 | __pre_bwrite($e_s$) __post_bread($e_l$) | 4 |
| ... | ... | ... | ... | ... |
| 413175 | Overall | 1.24 | Overall | 4.35 |

**Table 3: Buffer and primitive annotation counts and complexity (C)**

as quality gates, such that by certain milestones teams must annotate all buffers matching certain criteria. In total, over 400,000 annotations have been added to source code for the product.

While the inference is run centrally to generate annotations, control over annotation insertion is given to individual developers. A tool fetches inferred annotations from a central location, and offers them for (optional) inspection before inserting them. Through informal surveys, we estimate that over 150,000 annotations were inserted using this tool.

The checker is deployed on the desktop machine of every developer working on the product, finding overflows in code within minutes of being written. All warnings are available to developers, and all level 1 warnings (Section 5.1) must be fixed before check-in. In total, over 3,000 have been fixed due to the checker.

## 6.1 Language Effectiveness

By using separate primitive and buffer annotation layers, we seek to annotate the great majority of buffer interfaces concisely, but still have the expressive power to annotate the many unusual interfaces that may be in use.

To gauge the annotations' effectiveness at concisely representing interfaces, we check the relationship between the frequency of an interface and the complexity of the annotation required to represent it. The most common interfaces should have the simplest representation. Table 3 relates interface frequency with the complexity of the corresponding buffer and primitive annotations for over 400,000 annotations. We measure the complexity of a set of annotations for an interface as the total number of values used from Table 1 or Table 2.

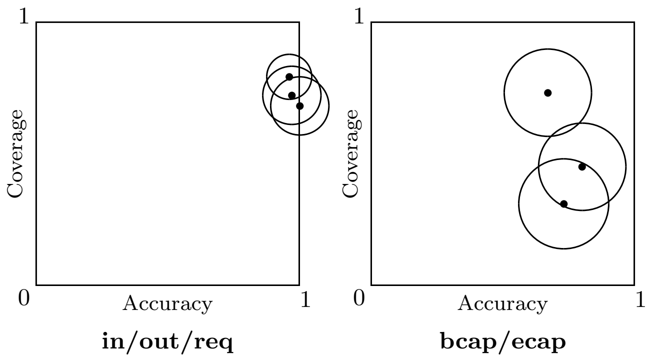
Weighted by frequency, interfaces are annotated with buffer annotations using an average of 1.24 values, or with primitive annotations using an average of 4.35 values, a 251% increase. Additionally, as can be seen from the annotation lengths in Table 3, the least complex buffer annotations describe the most common interfaces, while there is little correlation between primitive annotation complexity and interface frequency.

## 6.2 Inference Effectiveness

Running inference specifications over the entire product's code bases yields ~1,800,000 `in`, `out`, and `req` annotations, and ~120,000 `ecap` and `bcap` annotations. We use sampling to gauge the accuracy and coverage of these annotations, and thus their overall quality and effectiveness. Accuracy is estimated as the fraction of a random sample of inferred annotations which are correct, and coverage is estimated as the fraction of a random sample of code base function parameters for which the correct annotations were inferred. Table 4 shows the result of this sampling for three code bases totalling 9.8 MLOC. Accuracy and coverage are listed and plotted with 95% confidence intervals; `in`, `out`, and `req` have larger samples and thus tighter confidence intervals.

For `in`, `out` and `req`, we estimate that we infer 75% of the correct annotations for a code base with a 3% false positive rate. Most false negatives are due to separately analyzing code bases which may call into each other, as parameter information is not propagated across such calls. The few false positives are due to aliasing; for example, if one alias for a value is checked against `NULL` before another alias is dereferenced, a `req` annotation will still be inferred.

For `ecap` and `bcap`, we estimate that we infer 49% of the correct annotations for a code base with a 28% false positive rate. Most false negatives are due to complex pointer and size arithmetic that is not handled by the specification, isolated functions to which sizes cannot be propagated, and, as with `in`, `out` and `req`, separately analyzing code bases that call into each another. Most false positives are due to confusion between byte and element counts on a buffer, or between multiple buffers and multiple sizes; for example, if two buffers are passed to a function with identical constant sizes, those buffer/size relationships will be tangled together. In practice, we exploit type and naming conventions to reduce the false positive rate to less than 10%.



| | in/out/req | | bcap/ecap | |
|---|---|---|---|---|
| MLOC | Accuracy | Coverage | Accuracy | Coverage |
| 2.0 | $1.0 \pm .11$ | $.68 \pm .11$ | $.80 \pm .15$ | $.45 \pm .18$ |
| 6.1 | $.96 \pm .08$ | $.79 \pm .09$ | $.73 \pm .16$ | $.31 \pm .18$ |
| 1.7 | $.97 \pm .12$ | $.72 \pm .10$ | $.67 \pm .17$ | $.73 \pm .16$ |
| 9.8 | $.97 \pm .04$ | $.75 \pm .06$ | $.72 \pm .09$ | $.49 \pm .11$ |

**Table 4: Inference results for `in/out/req` and `bcap/ecap`**

## 6.3 Checker Effectiveness

We have tested the checker on the suite of buffer overflow model programs developed by Zitser et. al. [18]. These are based directly on vulnerable and patched versions of several open source server applications, and form the most realistic publically available testing suite of which we are aware. Of the 14 vulnerable programs, 5 may overflow due to the incorrect use of unsafe string functions (e.g. `strcpy` and `sprintf`). Uses of these functions are identified and removed through a separate checking process, and as such we do not annotate or analyze them (see Section 3.2). Of the remaining 9 programs, we supplied relevant annotations (all of which are on standard library functions or are automatically inferred) and ran our checker, which issued warnings for the relevant lines in 7 vulnerable programs (true positives) and 3 patched programs (false positives). Most warnings are level 2 and 3. The results of these runs are shown in Table 5.

Of the two missed overflows, SM-6 is on a global buffer which we do not yet annotate (see Section 5.1), and BIND-2 is due to a signed-to-unsigned cast of a potentially negative size variable, a different type of bug which is caught by a separate integer overflow checker not described here.

Figure 7 shows checker performance on numerous partially annotated code bases (all overflows resulting from these warnings have been fixed). Warning counts are moderately correlated with annotation density, with an $R^2$ measure of predictive power of 42%. Overflows are readily found on unannotated code, with coverage steadily increasing as annotations are incrementally added.

## 7. RELATED WORK

Many annotation-based checkers have been developed to detect buffer overflows and other security errors, including CSSV [5], Eau Claire [2], Splint [7], and MECA [17]. The principal novelty with our system is our incremental approach to annotation. Existing systems either require fully annotated code bases to perform effective checking [5, 2] or cannot comprehensively check even fully annotated code bases [7, 17].

| Name | True Pos. | False Pos. | Annotations |
|---|---|---|---|
| SM-1 | ✓ | | |
| SM-3 | ✓ | ✓ | 1 inferred |
| SM-4 | ✓ | ✓ | |
| SM-5 | ✓ | | 1 inferred |
| SM-6 | | | |
| SM-7 | ✓ | ✓ | 2 library |
| BIND-1 | ✓ | | 1 library |
| BIND-2 | | | 1 library |
| BIND-3 | ✓ | | 1 library |

**Table 5: Checker results on Zitser model programs**

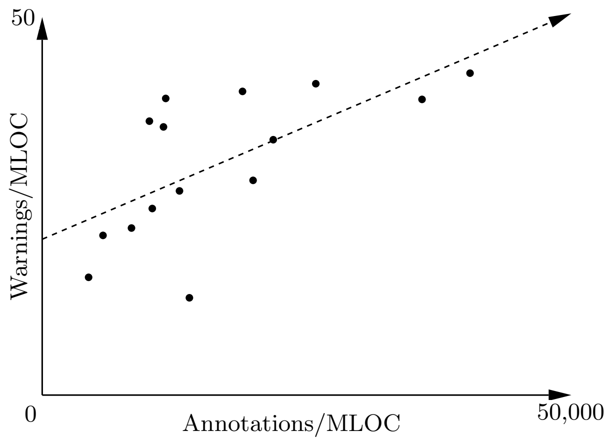


**Figure 7: Annotation and level 1 warning rates**

CSSV [5] is a modular overflow checker for string-intensive C programs. CSSV's annotation language is considerably more expressive but considerably less concise than ours. Sound contracts are inferred by approximating the weakest preconditions and strongest postconditions on functions, but lack coverage compared to manual annotation. Functions are combined with contracts to produce an integer program which is checked for violations. This method is sound and precise, but requires the analyzed function and its callees to be fully annotated.

ESC/Java [9] uses a sophisticated theorem prover for general purpose modular checking. Eau Claire [2] uses ESC to find buffer overflows in C programs, extending the annotation language with constructs similar to that of CSSV. Since the checking method used by ESC does not widen loops, either invariants must be provided or a fixed number of iterations are simulated, a limited approach for analyzing loop-intensive buffer manipulations.

Splint [7] extends LCLint with annotations for buffer readable and writable extents. Fairly complex annotations are necessary even for simple buffer/size relationships, and no inference mechanism is available. Splint has an unsound constraint-based checker which uses heuristics to analyze common loop forms. The effectiveness of this strategy is unclear, though in [7] the false positive rate was 75% after annotation.

MECA [17] is an annotation-based checker built on MC [1]. An initial set of annotations provided by the programmer is propagated bottom-up through the call graph, after which statistical inference is used to select likely additional annotations for confirmation by the programmer. Our inference is more aggressive, propagating information top-down without being checked. MC can be used to find operations leading to overflows, such as unchecked array accesses [1], but not comprehensively check for them.

Houdini [8] infers annotations by using predefined heuristics to guess potential annotations, including buffer/size relationships, and eliminating ones refuted by ESC/Java. Houdini achieves high accuracy and coverage, but due to the number of guesses required and expense in refuting them, seems to scale poorly.

Daikon [6] infers annotations by running an instrumented program through a test suite and collecting invariants. Such a dynamic approach may generate invariants that hold over the test cases but not over all inputs. As with Houdini, Daikon can be used in conjunction with ESC/Java [10] to remove incorrect invariants.

Static buffer overflow checkers which do not rely on annotations can be deployed on code bases with little risk. These include ARCHER [16], BOON [15], and the PolySpace C Verifier [11]. While offering great utility for finding overflows, the requirement for very precise interprocedural analysis limits the effectiveness of these checkers. Zitser et. al. test these and other checkers on Sendmail, a 145,000 line program, and 14 model programs [18]. ARCHER and BOON can analyze programs the size of Sendmail, but find only 1 and 2 of the model program overflows, respectively. In contrast, the PolySpace C Verifier finds almost all of the model program overflows, but cannot analyze Sendmail. Our checker achieves coverage similar to that of the PolySpace C Verifier (Section 6.3) while analyzing programs of any size, at the expense of requiring annotations.

Dynamic overflow mitigation techniques include checking stack integrity to detect overflows after they occur [4], or checking memory accesses to prevent overflows from occurring [3, 14, 13]. Overhead generally increases with the amount of protection provided. Static and dynamic techniques are complementary, with static checkers finding many overflows before runtime, and dynamic checkers finding those that get through.

## 8. CONCLUSION

Modular checking is a very effective way to find and check the absence of buffer overflows. Ensuring memory safety is only one of many steps needed to secure a code base. Many additional static checkers are currently used throughout the development process to find additional buffer overflows, integer overflows, `NULL` dereferences, uses of uninitialized memory, uses of tainted data, and numerous product-specific security errors. To various degrees, all of these checkers leverage the buffer annotations for better results; the value provided by annotations constantly increases.

## REFERENCES

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes, May 2002. In IEEE Symposium on Security and Privacy, Oakland, California.

[2] B. Chess. Improving computer security using extended static checking. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 160, Washington, DC, USA, 2002. IEEE Computer Society.

[3] J. Condit, M. Harren, S. McPeak, G. Necula, and W. Weimer. CCured in the real world. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 232–244, New York, NY, USA, 2003.

[4] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[5] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 155–167, New York, NY, USA, 2003. ACM Press.

[6] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions in Software Engineering*, 27(2):99–123, February 2001.

[7] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, 2002.

[8] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *Symposium of Formal Methods Europe*, 2001.

[9] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, 2002.

[10] J. Nimmer and M. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*, 2001.

[11] Polyspace - automatic detection of run-time errors at compile time. www.polyspace.com.

[12] T. Reps, M. Sagiv, and S. Horwitz. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages*, 1995.

[13] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and W. Beebee Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[14] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pages 159–169, 2004.

[15] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[16] Y. Xie, A. Chou, and D. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *FSE '03: Proceedings of the 10th ACM SIGSOFT international symbosium on Foundations of software engineering*, 2003.

[17] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: an extensible, expressive system and language for statically checking security properties. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

[18] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106, New York, NY, USA, 2004. ACM Press.