Software Engineering Institute

COVERT: A Framework for Finding Buffer Overflows in C Programs via Software Verification

Sagar Chaki
Arie Gurfinkel

August 2010

TECHNICAL REPORT
CMU/SEI-2010-TR-029
ESC-TR-2010-029

Research, Technology, and System Solutions
Unlimited distribution subject to the copyright.

http://www.sei.cmu.edu

Carnegie Mellon

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Copyright 2010 by Carnegie Mellon University.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Internal use. Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and “No Warranty” statements are included with all reproductions and derivative works.

External use. This document may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other external and/or commercial use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about SEI publications, please visit the library on the SEI website (www.sei.cmu.edu/library).

Table of Contents

Abstract .... **vii**

1 Introduction **1**

2 Related Work **5**

3 Instrumentation **7**
- 3.1 Memory Model 7
- 3.2 Basic Instrumentation 7
- 3.3 Handling Memory Allocation and Standard String Manipulation Routines 9
  - 3.3.1 Malloc 10
  - 3.3.2 Strcpy 10
  - 3.3.3 Strcat 11
- 3.4 Discussion 11
  - 3.4.1 Checking Underflow 11
  - 3.4.2 Entry Points for Analysis Engine 11
  - 3.4.3 Aliasing 11
  - 3.4.4 Aiding Numerical Analysis 11
  - 3.4.5 Other Dynamic Approaches to Buffer Overflows 11

4 Analysis **13**
- 4.1 Assumptions About CHECKER 13
- 4.2 PANA: Combining Numeric and Predicate Abstraction 13
- 4.3 Numeric Abstraction 13
- 4.4 Predicate Abstraction 14
- 4.5 Combining Numeric and Predicate Abstractions 14
- 4.6 Abstraction Refinement 15

5 Implementation and Validation **17**
- 5.1 Experimental Validation 17

6 Conclusion **19**

References **21**

List of Figures

Figure 1: Using COVERT with Static (SA) and Dynamic (DYN) Buffer Overflow Prevention Techniques 3

Figure 2: A Memory with a Character Buffer and Two Pointers 8

Figure 3: (a) COVERT's Fat Pointer Structure (b) Macros Used by INSTRUMENTATION 8

Figure 4: Assignment in (a) Is Replaced by Template in (b) during INSTRUMENTATION. 9

Figure 5: The Code Fragment (a) Is Converted to the Code Fragment (b) by INSTRUMENTATION. 9

Figure 6: Definition of `fp_malloc` 10

Figure 7: Definition of `fp_strcpy_alarm` 10

Figure 8: Definition of `fp_strcat_alarm` 10

Figure 9: A Code Fragment (a) and Its Instrumentation (b) 15

Figure 10: Bar Chart Showing Comparison of Running Times Between COPPER, BLAST, and PANA as the CHECKER for All Tests 18

List of Tables

Table 1: Common Abstract Domains . . . 14
Table 2: Summary of implementations of NUMPREDDOM . . . 15
Table 3: BLAST, COPPER, and PANA Comparison . . . 17

Abstract

Buffer overflows continue to be the source of a vast majority of software vulnerabilities. Solutions based on runtime checks incur performance overhead, and are inappropriate for safety-critical and mission-critical systems requiring static—that is, prior to deployment—guarantees. Thus, finding overflows statically and effectively remains an important challenge. This report presents COVERT, an automated application framework aimed at finding buffer overflows in C programs using state-of-the-art software verification tools and techniques. Broadly, COVERT works in two phases: INSTRUMENTATION and ANALYSIS. The INSTRUMENTATION phase is the core phase of COVERT. During INSTRUMENTATION, the target C program is instrumented such that buffer overflows are transformed to assertion violations. In the ANALYSIS phase, a static software verification tool is used to check for assertion violations in the instrumented code, and to generate error reports. COVERT was implemented and then evaluated on a set of benchmarks derived from real programs. For the ANALYSIS phase, experiments were conducted with three software verification tools—BLAST, COPPER, and PANA. Results indicate that the COVERT framework is effective in reducing the number of false warnings, while remaining scalable.

1 Introduction

A 2002 study funded by the National Institute of Standards and Technology (NIST) concluded that software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated $60 billion annually [NIST 2002]. A substantial portion of programming errors ultimately manifest themselves as software vulnerabilities. For example, it is estimated that "Hacker attacks cost the world economy a whopping $1.6 trillion in 2000" and that "U.S. virus and worm attacks cost $10.7 billion in the first three quarters of 2001" [Jarzombek 2004]. This problem is further highlighted by the increasing number of successful attacks. For example, "The CMU CERT® Coordination Center reported 76,404 attack incidents in the first half of 2003, approaching the total of 82,094 for all of 2002 in which the incident count was nearly four times the 2000 total." In fact, CERT statistics often understate the problem by counting all related attacks as a single incident.

Buffer overflows are widely recognized to be the prime source of vulnerabilities in commodity software [Cowan 2000]. For example, the CodeRed worm that caused an estimated global damage worth $2.1 billion in 2001 [Jarzombek 2004, CERT CC 2001] exploited a buffer overflow in Windows. Wagner and colleagues report, on the basis of CERT advisories, that "buffer overruns account for up to 50% of today's vulnerabilities, and this ratio seems to be increasing over time" [Wagner 2000]. A recent SANS/MITRE study cited buffer overflows as one of the top 25 most dangerous programming errors [MITRE-CWE-09 2009].

Buffer overflows are problematic because they are used by attackers to execute arbitrary code (such as a shell) with administrative privileges. For example, a common strategy is to redirect a program's control flow to any desired point by overflowing buffers. For this reason, buffer overflows are extremely dangerous and can lead to catastrophic system compromises and failures.

Broadly speaking, a buffer overflow occurs when some data D is written to a buffer B and the size of D is greater than the allocated size of B. In the case of a type-safe language or a language with runtime bounds checking (such as Java), an overflow leads either to a (compile-time) type error or a (runtime) exception. In such languages, a buffer overflow can lead to a denial of service attack (i.e., by causing an unhandled exception), but in most cases cannot be used to compromise the security of the system. Unfortunately, a significant fraction of current and legacy software is written in unsafe languages (such as C or C++) that allow buffers to be overflowed with impunity. For reasons such as efficiency and infrastructural inertia, the unsafe use of these languages is unlikely to abate. Note that the overflow problem is not solved by restricting the programmers to using only the "safer" library routines, such as `fgets`, `snprintf`, and `strncpy`, because programmers can, and do, pass incorrect array bounds information to these routines. Therefore, it is important to develop techniques to guard against buffer overflows, while still allowing low-level buffer accesses.

A number of *static* and *dynamic* approaches have already been used effectively to partially mitigate the buffer overflow problem. Dynamic approaches (e.g., by Ruwase, Jones, Dahn, and Dhurjati) work by instrumenting buffer accesses of the program with runtime checks that abort the program as soon as a buffer overflow is detected [Ruwase 2004, Jones 1997, Dahn 2003, Dhurjati 2006]. The approaches differ in whether the instrumentation is done on the source or binary levels, in runtime and memory overheads, and in compatibility with third-party library routines that cannot be instrumented. The runtime overhead is the major cost of a dynamic approach, ranging anywhere from a 2x to 10x slowdown. This often defeats the performance advantages of using an unsafe low-level language. Moreover, by aborting a program when an overflow is detected at runtime, dynamic approaches often eliminate overflows at the cost of introducing denial-of-service attacks. This is unacceptable in many situations where downtime is extremely expensive, for example, finance, telecommunication, and avionics.

Static approaches work by examining the source code of the program statically, looking for conditions (i.e., program inputs) that may result in a buffer overflow. Several such techniques (e.g., those of Wagner and

® CERT Coordination Center is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

Ganapathy) have been used to find buffer overflows in industrial-scale software [Wagner 2000], [Ganapathy 2003]. While these approaches are very scalable, in many situations they produce a large number of warnings (or alarms). These warnings must then be inspected manually, which makes the overall process very tedious. Alternatively, the warnings can be guarded by runtime checks, but this compromises the strong guarantees of static analysis, since it introduces the prospect of a runtime abort. In practice, many of these warnings turn out to be false alarms—owing to the *imprecision* that static analysis tools must allow for in order to achieve scalability. For example, Zitser and colleagues report false alarm rates of as high as 50% when using static analysis tools for buffer overflow detection in real programs [Zitser 2004]. Today an effective use of static analysis for buffer overflows requires a significant manual effort from the user—either in manually examining false alarms, or in guiding a static analysis tool by annotating the program with tool-specific annotations.

On the other hand, there is a wide array of existing software verification tools—for example, BLAST [Henzinger 2002], CBMC [Clarke 2004], COPPER [Chaki 2005], and PANA [Gurfinkel 2008]—that use counter-example guided abstraction refinement (CEGAR) to eliminate or minimize false positives [Clarke 2003, Ball 2001]. These tools generally detect assertion violations (or equivalently, reachability of a statement) in C programs. Thus, in principle, they are applicable to detecting buffer overflows. However, in practice, there is no mechanism to do this systematically.

In this report, we present an automated framework[1], called COVERT, that provides such a mechanism. The input to COVERT is a pair (P, ALARM) of a program P, and a set ALARM of control locations of P with possible buffer overflows. We assume that the set ALARM is either generated by some other static analysis technique, or simply contains all control locations of P. The output from COVERT is a triple $(\text{GOOD}, \text{BAD}, \text{ALARM}')$ such that

- $\text{GOOD} \subseteq \text{ALARM}$ is a list of *control locations* of P that are free of buffer overflows.
- BAD is a list of *execution traces* of P leading to buffer overflows.
- ALARM' is the list of *control locations* for which the technique failed to prove safety and failed to construct an execution leading to a buffer overflow.

Thus, the program P has no buffer overflows if both ALARM' and BAD are empty, has a demonstrable buffer overflow if BAD is not empty, and has potential buffer overflows if ALARM' is not empty. The current implementation of COVERT looks only for buffer overflow in C strings (i.e., null-terminated arrays of characters), but the techniques easily generalize to arbitrary buffer overflows as well.

COVERT works in two stages, called INSTRUMENTATION and ANALYSIS. The INSTRUMENTATION phase converts the target program P into a new program P_i. COVERT's instrumentation is *sound*. In other words, any input that leads to a buffer overflow at some ALARM location in P causes an assertion violation at the corresponding location in P_i. This is similar to what is traditionally done in dynamic approaches to buffer overflows. The key difference is that our instrumentation is targeted towards efficient static analysis and not towards efficient runtime behavior. The instrumentation replaces each character array (`char*`) reference in P with a specialized "fat" pointer that keeps track of the size of the buffer and the length of the string contained in it, and adds assertions to check whether the buffer overflows.

In the ANALYSIS phase, COVERT uses a safety analysis engine, henceforth called CHECKER, to check for possible assertion violations in P_i. If an assertion in P_i is proved to be safe, the corresponding ALARM location in P is added to GOOD. If an execution trace leading to a violation of an assertion in P_i is discovered, then the corresponding execution trace in P is added to BAD. If a control location in ALARM could not be classified as either GOOD or BAD, it is added to ALARM'. It is noteworthy that even though COVERT uses code instrumentation, it is a static technique—that is, it is applicable prior to deployment—and hence does not suffer from the disadvantages of dynamic analysis techniques.

---

[1] Specifically, we present an application framework.

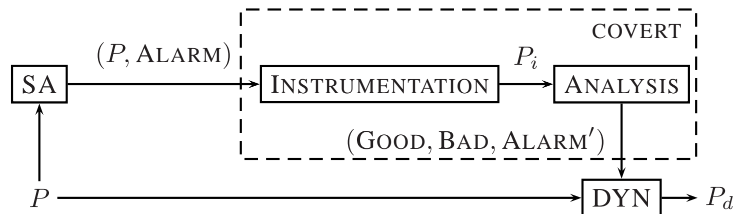

Figure 1: Using COVERT with Static (SA) and Dynamic (DYN) Buffer Overflow Prevention Techniques

One of the strengths of COVERT is that it allows CHECKER to be instantiated by a wide variety of software verification tools, subject to some reasonable assumptions (see Section 4.1 for a discussion on these assumptions). This enables COVERT to leverage the state of the art in software verification technology. In particular, we experimented with the following three instantiations of CHECKER: (1) the software model checker BLAST [Henzinger 2002], (2) the software model checker COPPER [Chaki 2005], and (3) PANA [Gurfinkel 2008], a software verification tool that combines three sophisticated techniques: numeric abstraction [Miné 2006], predicate abstraction [Graf 1997], and CEGAR. Further details about the ANALYSIS phase of COVERT are presented in Chapter 4.

Another strength of COVERT is that it can be combined naturally with other static and dynamic techniques for buffer overflow prevention. A typical work-flow is illustrated in Figure 1. In the figure, P is a program, P_i is the program P instrumented for analysis, and P_d is the program P instrumented with runtime checks.

First, the set of possible buffer overflow locations is identified by a scalable (but conservative) static analysis tool SA. Second, COVERT is used to eliminate false alarms, and, if possible, produce error traces for real buffer overflows. Third, any of the alarms that could not be conclusively classified are protected with dynamic runtime checks.

We implemented and evaluated COVERT on a set of benchmarks derived from real C programs with buffer overflows for which existing static analysis tools are known to be inadequate [Zitser 2004]. We compared between the three CHECKER instances—BLAST, COPPER, and PANA. Our experiments indicate that PANA is superior to both BLAST and COPPER in terms of successfully, and quickly, proving the presence or absence of buffer overflows in realistic C programs.

The rest of this report is organized as follows. In Section 2 we survey related work. In Section 3 and Section 4 we describe the instrumentation and analysis stages of COVERT, respectively. In Section 5, we present our implementation and evaluation of COVERT. Finally, we conclude in Section 6.

2 Related Work

Manual approaches for overflow detection are inherently non-scalable, therefore we focus on automated procedures only. A number of approaches for overflow detection are type-theoretic [Shankar 2001] in nature. These approaches require that programs be written in a type-safe language and are not applicable to the vast body of (legacy as well as in-production) code that involves type-unsafe languages such as C or C++. Techniques based on simulation or testing suffer from low coverage and are typically unable to provide any reasonable degrees of assurance about critical software systems. Dynamic or runtime buffer overflow detection schemes [Ruwase 2004, Jones 1997, Dahn 2003, Dhurjati 2006] incur performance penalties that are unacceptable in many situations. Even when performance is not a serious issue, it is often imperative that we be assured of the correctness of a system before it is deployed. Such guarantees can only be obtained, if at all, via static approaches.

A number of static approaches for buffer overflow detection have been proposed that rely on static analysis of programs. These approaches are usually based on converting the buffer overflow problem into a constraint solving problem, such as integer range checking [Wagner 2000] or integer linear programming [Ganapathy 2003], or to a static analysis problem on an integer program [Dor 2003]. Static analysis amounts, in principle, to a form of model checking over the control flow graph (CFG) of a program [Schmidt 1998]. However, a CFG is an extremely imprecise model because it retains control flow information but ignores other semantic details completely. Thus, in practice, static analysis based on the CFG is plagued by false alarms [Zitser 2004]. COVERT uses abstraction refinement to eliminate false alarms in an automated manner.

Hovemeyer and colleagues use an unsound and incomplete static analysis to find NULL pointer bugs [Hovemeyer 2005]. Beyer and colleagues use BLAST to also check for NULL pointer bugs in C programs [Beyer 2005]. Specifically, they use BLAST to detect violations of runtime checks inserted by the CCURED tool. However, the checks added by CCURED are geared toward preserving the runtime behavior of the target program [Necula 2005]. They involve complicated pointer manipulations and dynamic memory allocation, and therefore are not easy to analyze statically for a more general class of memory errors, for example, buffer overflows. In contrast, the runtime checks used by COVERT limit dynamic memory allocation and pointer dereferencing, are designed to be static-analysis friendly, and are targeted toward buffer overflows.

3 Instrumentation

The instrumentation stage of COVERT transforms a C program P and a set of locations ALARM into a new C program P_i by adding for every location Loc in ALARM a set of assertions ASSERT(Loc) such that the following claim holds:

Claim 1 (Soundness) *Any input that leads to a buffer overflow at a location Loc in* ALARM *in P also causes a violation of an assertion in* ASSERT(Loc) *in P_i. If P has no buffer overflows, P_i behaves exactly the same as P.*

That is, buffer overflows in the original program P are reduced to assertion violations in the instrumented program P_i. COVERT's instrumentation phase is really a form of a dynamic approach of adding runtime checks (or assertions) to prevent buffer overflow. The key difference from other dynamic approaches is that the meta-data and assertions used by COVERT are designed not for runtime performance, but to be easily checkable via static analysis.

In the rest of this section, we describe the instrumentation process and its data structures, and explain how COVERT handles string manipulation functions from the standard C library. We conclude with a discussion of our approach.

3.1 Memory Model

COVERT divides memory into two regions: a region for character buffers, and a region for all other data. The memory for character buffers is modeled as a set of disjoint objects—one per buffer. Each object is identified by its base address, and has two properties: (i) allocated size, and (ii) string length, which is the position of the first `'\0'` character if one exists, and the allocated size otherwise. A pointer `p` to a character buffer `B` is represented as an integer offset from the base address of the memory object that contains `B`. We say that `B` is an intended referent (IR) of `p`.

For example, Figure 2 shows a memory with a single allocated object O, and two pointers `p` and `q` with O as their IR. The base address of O is 10, the size is 7, and string length is 5. The offset of `p` is 2, so that `p` is "`llo`," and the offset of `q` is 4, so that `q` is "`o`."

3.2 Basic Instrumentation

COVERT implements the above memory model by using "fat pointers" to keep track of meta-data, such as allocated size, with each pointer. This technique is used in a wide variety of applications ranging from memory management and memory profiling to dynamic analysis [Dhurjati 2006] and ensuring memory safety [Necula 2005]. The key idea of fat pointers is to replace each pointer variable p (or each `char*` variable in COVERT's case) with a data structure that keeps track of at least: (1) the actual address that p contains, and (2) the base address of the block of memory that p points into, also known as the *intended referent* of p. In addition, a COVERT fat pointer also keeps track of the *size* of the intended referent of p. Specifically, a COVERT fat pointer is declared by the structure `fp_char` shown in Figure 3(a). The fields of the structure are

1. `base`: base address of the intended referent
2. `offset` the offset from `base` to `data`, that is, `offset` = `data` − `base`
3. `alloc` the allocated size of the intended referent
4. `len` a *pointer* to the *string length* of the intended referent, that is, the offset of the *first* `NULL` character from `base` if one exists, or `alloc` otherwise

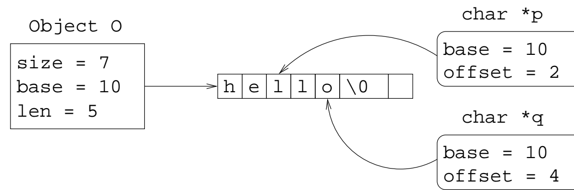

Figure 2: A Memory with a Character Buffer and Two Pointers

```
struct fp_char {
   char *base ;
   int offset ;
   int alloc ;
   int *len ;
};
```

(a)

```
#define FPDATA(p) (p.base + p.offset)
#define IN_BOUNDS(p) (p.offset < p.alloc)
#define NULL_TERM(p) (p.offset <= *(p.len))
#define STRLEN(p) (*(p.len) - p.offset)
#define MIN(x,y) ((x < y) ? x : y)

extern int GET_LEN(fp_char x);
```

(b)

Figure 3: (a) COVERT*'s Fat Pointer Structure (b) Macros Used by* INSTRUMENTATION

The `len` field is maintained as a pointer to enable us to model aliasing (see Section 3.4.3 for more details). If `len` points to a value between zero and `alloc - 1`, then the intended referent is null-terminated; otherwise, the intended referent is not null-terminated. The meta-data captured by the fields of `fp_char` lets us model string manipulations performed by the program via simple numeric operations.

Recall that the INSTRUMENTATION phase takes a C program P as input and produces an instrumented C program P_i. In addition to functions and macros used by P, P_i may use macros and external functions shown in Figure 3(b). In the macros, `p` is assumed to be of type `fp_char`. The macro `IN_BOUNDS(p)` checks whether `p` overflows its intended referent, `NULL_TERM(p)` checks if `p` is null-terminated, and `STRLEN(p)` returns the computed *string length* of `p`. The function `GET_LEN(p)` computes the exact string length of `p.base`. Specifically, it returns the offset, from `p.base`, of the first `NULL` character between `p.base` and `p.base + p.alloc - 1`. If so such `NULL` character exists, it returns `p.alloc`. The INSTRUMENTATION phase proceeds as follows:

1. P is reduced to use only simple types, variables, and statements. For example, complicated expressions (such as those involving nested function calls, `++` and `--` operators, etc.) are broken down into simpler form.

2. The type `char*` is promoted to type `fp_char` throughout P. Other types are left unchanged.

3. For each string constant `c`, a unique `fp_char` variable `v` is created and initialized appropriately. All subsequent references to `c` are replaced by `v`.

4. Any variable `x` of type `char*` is replaced by `FPDATA(x)` if
   - `x` is a sub-expression of a branch condition, a function argument, or a return expression, or
   - `x` is a sub-expression of the right-hand-side of an assignment whose left-hand-side is *not* of type `fp_char`.

5. Every assignment of the form `l = e`, where `l` is of type `fp_char` is converted to a sequence of assignments represented by the template shown in Figure 4. The exact

definitions of `BASE(e)`, `OFFSET(e)`, `ALLOC(e)` and `LEN(e)` depends on the structure of `e`. These definitions are straightforward, and we omit a detailed presentation for brevity.

6. Every assignment of the form `*l = e`, where `l` is of type `fp_char` is preceded by the following code fragment, which updates the length of `l` if required.

   ```
   *(l.len) = (e == '\0') ?  MIN(l.offset,*(l.len)) : *(l.len);
   ```

7. Every assignment in ALARM of the form `*l = e`, where `l` is of type `fp_char`, is preceded with a statement `assert(IN_BOUNDS(l))`. Note that this `assert` statement follows the instrumentation introduced by the previous step.

8. Some specific function calls are handled in a special way, as described in the next section.

(a)

```
l = e;
```

(b)

```
l.base = BASE(e);
l.offset = OFFSET(e);
l.alloc = ALLOC(e);
l.len = LEN(e);
```

Figure 4: Assignment in (a) Is Replaced by Template in (b) during INSTRUMENTATION.

(a)

```
char *p="abc";

if(p[0]=='a')
{
  p++;
}
```

(b)

```
1: fp_char t;
2: t.base="abc";
3: t.offset=0;
4: t.size=4;
5: t.len=malloc(ISZ);
6: *(t.len)=3;
7: fp_char p;
8: p.base=t.base;
9: p.offset=t.offset;
10:p.size=t.size;
11:p.len=t.len;
12:if(FPDATA(p)[0]=='a') {
13:  p.base=p.base;
14:  p.offset++;
15:  p.size=p.size;
16:  p.len=p.len;
17:}
```

Figure 5: The Code Fragment (a) Is Converted to the Code Fragment (b) by INSTRUMENTATION.

To illustrate INSTRUMENTATION, consider code fragments before and after INSTRUMENTATION as shown in Figure 5(a) and Figure 5(b), respectively. In the code, `ISZ` stands for `sizeof int`, and variable `t` stands for the string constant `"abc"`. In the instrumented code, lines 1–6 initialize `t`, lines 7–11 are the assignment to `*p`, line 12 is the branch conditional, and lines 13–17 are the increment of `p`. Note that we have expanded `BASE`, `OFFSET`, `SIZE`, and `LEN` for a pointer increment in this example (see lines 13, 14, 15 and 16, respectively).

3.3 Handling Memory Allocation and Standard String Manipulation Routines

Effective static analysis crucially depends on partitioning the analysis problem across function boundaries. To this end, we have enhanced COVERT's fat-pointer instrumentation to model the semantics of common string manipulation routines. That is, calls to such functions as `strcpy` and `strcat` are instrumented so that the `fp_char` meta-data is updated directly based on the semantics of these functions. In this section, we describe the instrumentation for `malloc`, `strcpy`, and `strcat`. The instrumentation for other functions is done similarly.

```
fp_char fp_malloc(size_t e) {
1:   fp_char p;
2:   p.base = (char*)malloc(e);
3:   p.offset = 0;
4:   p.alloc = e;
5:   p.len = malloc(sizeof int);
6:   *(p.len) = GET_LEN(p);
7:   return p;
}
```

Figure 6: Definition of `fp_malloc`

3.3.1 Malloc

A call to `l = malloc(e)`, where `l` is of type `fp_char`, is replaced by a call to `l = fp_malloc(e)`. The function `fp_malloc` is defined in Figure 6. The function allocates the space for the pointer, sets allocation size and offset meta data, and (re)computes the string length of the allocated memory block to ensure that the length is initialized appropriately. Note that this is necessary since we don't know the contents of the newly allocated block of memory in advance.

```
fp_char fp_strcpy_alarm(fp_char x,fp_char y) {
1:   assert(NULL_TERM(y));
2:   assert(x.offset + STRLEN(y) < x.alloc);
3:   strcpy(FPDATA(x),FPDATA(y));
4:   *(x.len) = NULL_TERM(y) ?
5:       x.offset + STRLEN(y) : GET_LEN(x);
6:   return x;
}
```

Figure 7: Definition of `fp_strcpy_alarm`

3.3.2 Strcpy

A call to `strcpy(x,y)` is replaced by a call to `fp_strcpy_alarm(x,y)` or to `fp_strcpy_no_alarm(x,y)` depending on whether the original call was in the ALARM set or not. The definition of `fp_strcpy_alarm` is shown in Figure 7. Lines 1–2 check for buffer overflows, line 3 makes the actual call to `strcpy`, lines 4–5 update the value of `*(x.len)`. Note that `GET_LEN` is called if the C expression for new value of `*(x.len)` cannot be determined statically. Finally, line 6 returns the result. The definition of `fp_strcpy_no_alarm` is the same as `fp_strcpy_alarm`, with lines 1 and 2 removed.

```
fp_char fp_strcat_alarm(fp_char x,fp_char y) {
1:   assert(NULL_TERM(x) && NULL_TERM(y));
2:   assert(*(x.len) + STRLEN(y) < x.alloc);
3:   strcat(FPDATA(x),FPDATA(y));
4:   *(x.len) = NULL_TERM(x) && NULL_TERM(y) ?
5:       *(x.len) + STRLEN(y) : GET_LEN(x);
6:   return x;
}
```

Figure 8: Definition of `fp_strcat_alarm`

3.3.3 Strcat

A call to `strcat(x,y)` is replaced by a call to `fp_strcat_alarm(x,y)` or to `fp_strcat_no_alarm(x,y)` depending on whether the original call was in the ALARM set or not. The definition of `fp_strcat_alarm` is given in Figure 8. Lines 1–2 check for buffer overflows, line 3 makes the actual call to `strcat`, lines 4–5 update the value of `*(x.len)`, and line 6 returns the result. The definition of `fp_strcat_no_alarm` is the same as `fp_strcat_alarm`, with lines 1 and 2 removed.

3.4 Discussion

In this section, we discuss a number of issues related to COVERT's fat pointer design.

3.4.1 Checking Underflow

The definition of `IN_BOUNDS(p)` in Figure 3(b) checks only for overflow. To check for buffer underflow, the macro is changed to `(p.offset >= 0)`. To check for both overflow and underflow, the macro is changed to:

```
((p.offset < p.alloc) && (p.offset >= 0))
```

3.4.2 Entry Points for Analysis Engine

An analysis engine used to validate P_i must decide on how to interpret `malloc`, `GET_LEN`, and standard C string manipulation routines (like `strcpy` and `strcat`). The simplest sound choice is to assume that these functions return a non-deterministic value and do not modify any memory that is not directly accessible through their arguments. This is the assumption we make during the ANALYSIS phase of COVERT. Other (existing and future) analyzers can model these functions differently, and this choice influences their precision versus scalability tradeoff.

3.4.3 Aliasing

The `len` field of `fp_char` is a *pointer* to an integer value. This ensures that fat pointers that have the same intended referent share the length field. Thus, updating this field through one particular fat pointer is reflected in all other fat pointers with the same intended referent. An alternative choice is to store the length with the `base` field. However, we believe our current design leads to a simpler fat pointer data structure.

3.4.4 Aiding Numerical Analysis

The instrumentation for `malloc`, `strcpy`, and `strcat` involved only simple numerical operations. We were able to instrument about two dozen commonly used string manipulation routines—`malloc`, `free`, `strcpy`, `strcpy_s`, `strncpy`, `strncpy_s`, `strcat`, `strcat_s`, `strncat`, `strncat_s`, `gets`, `gets_s`, `fgets`, `strlen`, `scanf`, `sprintf`, `snprintf`, `cuserid`, `getcwd`, `memset`, `memcpy`, `memcpy_s`, `memmove`, and `memmove_s`—using only such numerical operations. We believe that in most of these types of routines, the safety of a buffer access is provable, or a counterexample is deducible, by relying only on those numeric annotations. Our experience with different analysis engines (see Section 5) suggests that this is in fact the case.

3.4.5 Other Dynamic Approaches to Buffer Overflows

As we explained above, our INSTRUMENTATION phase can be seen as a form of dynamic runtime checks. In this domain, the approach of fat pointers is considered to have an unacceptable cost—for example, the runtime overhead, incompatibility with third-party library routines, and so on. An alternative solution is to maintain some information about each pointer (e.g., the allocated object a pointer belongs to) in a global data structure.

For example, Dhurjati and Adve use a global partitioned splay tree to keep track of all allocated memory objects [Dhurjati 2006]. So, checking for overflow is reduced to checking whether the memory address accessed belongs to an allocated memory object. Although such approaches perform well at runtime, we believe they are not well suited for producing instrumentation for static analysis. When used with static analysis, such an approach would require that, in addition to showing absence of buffer overflows, the analyzer establish the correctness of the global data structure (e.g., insertions, deletions, and lookups in the partitioned splay tree in the example above). Thus, they make an already difficult problem (detecting buffer overflows) even more difficult (proving correctness of data structures).

4 Analysis

Once the instrumented program P_i is generated, it is analyzed by COVERT's CHECKER. As mentioned previously, COVERT allows the CHECKER to be instantiated by any software verification tool that satisfies some reasonable assumptions, as we will discuss next.

4.1 Assumptions About CHECKER

The minimal requirements for CHECKER are that it

1. accepts input programs in the fragment of C with primitive data types (`int`, `char`, etc.), pointers, and structures
2. provides syntax to specify non-deterministic integer values
3. checks for assertion violations (or, equivalently, reachability of a program label)

The first requirement is obvious since we target C programs. Structures and pointers are used in the INSTRUMENTATION phase as explained earlier in Chapter 3. The second requirement ensures that CHECKER accepts non-deterministic models. For example, the syntax in BLAST is `x = __BLAST_NONDET` with the meaning that `x` is assigned a non-deterministic value. This requirement is essential for INSTRUMENTATION, especially for modeling the behavior of library routines soundly. We rely on the third requirement to convert buffer-overflows to assertion violations. Of course, the soundness of the overall analysis depends on the soundness of CHECKER with respect to the semantics of C.

As mentioned previously, we experimented with the following three instantiations of CHECKER: (1) BLAST, (2) COPPER, and (3) PANA [Gurfinkel 2008]. The BLAST [Henzinger 2002] and COPPER [Chaki 2005] software model checking tools use only predicate abstraction to construct models from programs [Graf 1997]. They are discussed elsewhere. In the rest of this section, we give an overview of PANA, which combines predicate abstraction and numeric abstraction for model extraction, and yielded the best experimental results.

4.2 PANA: Combining Numeric and Predicate Abstraction

Fundamentally, PANA works by combining two techniques called predicate abstraction and numeric abstraction. These two techniques statically infer program invariants in terms of the elements of an abstract domain. For any program P, the invariant at location Loc is an expression over P's variables that is true every time the execution of P reaches location Loc. For example, $x > 0$ is an invariant at location Loc if x is always positive whenever the program is at Loc. A program invariant is a map from every program location to the corresponding program invariants. A program location Loc is unreachable if there exists a program invariant that maps Loc to false.

Both numeric and predicate abstraction are instances of abstract interpretation [Cousot 1977]. They differ in the underlying abstract domain they use. Numeric abstraction is based on a numeric domain, such as Intervals [Cousot 1977] or Octagons [Miné 2006] over the numeric variables of the program. In contrast, predicate abstraction is based on a predicate domain, that is, the set of Boolean formulas over a finite set of predicates on program variables [Graf 1997].

4.3 Numeric Abstraction

Numeric abstraction uses a numeric domain to compute and represent program invariants. Three commonly used numeric abstract domains are shown in the top three rows of Table 1. In the table, V is a set of

Table 1: Common Abstract Domains

Name	Notation	Abstract Elements
Intervals	$\text{Box}(V)$	$\{c_1 \leq v \leq c_2 \mid c_1, c_2 \in \mathcal{N}, v \in V\}$
Octagons	$\text{Oct}(V)$	$\{\pm v_1 \pm v_2 \geq c \mid c \in \mathcal{N}, v_1, v_2 \in V\}$
Polyhedra	$\text{Pk}(V)$	linear inequalities over V
Predicates	$\text{Pred}(V)$	propositional formulas over V

numeric/propositional variables; $\mathcal{N}$ is the domain of numeric constants. For example, in the case of numeric abstraction with Octagons, a program invariant at every control location Loc is represented by an abstract value (an expression) of the form

$$\bigwedge \pm x \pm y \leq c \; ,$$

where x and y are numeric program variables, and c is a numeric (unbounded integral or real) constant. Thus, if the analysis concluded that at location Loc the invariant is $(x - y \leq 5) \wedge (y + z \leq -2)$, then whenever the program reaches Loc the difference between x and y is bounded by 5, and the sum of y and z is bounded by -2.

Note that numeric abstraction involves an infinite abstract domain (e.g., since c is an arbitrary constant) but abstract elements are of a restricted form. For example, arbitrary disjunctions and negations are not expressible in Octagons and have to be over-approximated. As the result, numeric abstractions tend to be very efficient and scalable, but their imprecision leads to a high rate of false alarms.

4.4 Predicate Abstraction

Predicate abstraction uses a Boolean formula over a finite set of predicates to compute and represent program invariants. For example, let a finite set of predicates $\mathcal{P}$ be defined as $\mathcal{P} = \{p, q\}$, where $p \equiv 2x + y < 0$ and $q \equiv x + z > 5$. Then, the abstract values available for predicate abstraction are p, q, $p \wedge q$, $p \vee q$, $p \wedge \neg q$, and so on. If the analysis declares that $\neg p \vee q$ is a program invariant at location Loc, then whenever the program execution reaches Loc, either $2x + y \geq 0$ or $x + z > 5$.

The advantage of predicate abstraction is that it supports predicates from any (semi)decidable theory, and allows for arbitrary propositional combination of predicates. Therefore, predicate abstraction is able to express a richer class of program invariants. It is able to detect a strong invariant in many situations where numeric abstraction fails with a false alarm. The disadvantage of predicate abstraction is its (lack of) scalability. Application of predicate abstraction requires an exponential (in the size of $\mathcal{P}$) number of calls to a (semi)decision-procedure for the theory from which the predicates are drawn.

4.5 Combining Numeric and Predicate Abstractions

PANA combines numeric and predicate abstraction to achieve both scalability and precision. The technical details of this approach are available elsewhere [Gurfinkel 2008]. The key idea is to use an abstract domain (called NUMPREDDOM) whose abstract elements are a combination of abstract elements from the numeric and predicate domains. A number of such combinations, with varying expressiveness and efficiency, are possible. The combinations supported by PANA are shown in Table 2. In the table, P = predicates; N = numerical abstract values; **Value** = type of an abstract element; **Example** = example of allowed abstract value; **Num** = numeric part representation (explicit or symbolic).

To illustrate the power of the combined domain, consider the code fragment shown in Figure 9(a), and its instrumented version shown in Figure 9(b). Using the combined domain, we can establish the following

program invariant at location L0:

$$(\texttt{cond} \wedge \texttt{x.alloc} = 5 \wedge \texttt{x.offset} = 0 \wedge \texttt{FPDATA(x)} = \texttt{FPDATA(p)}) \vee (\neg\, \texttt{cond} \wedge \texttt{x.alloc} = 8 \wedge \texttt{x.offset} = 0 \wedge \texttt{FPDATA(x)} = \texttt{FPDATA(q)})$$

This is sufficient to conclude that `(x.offset < x.alloc)` is a program invariant at locations `L1` and `L2`. Thus, there are no buffer overflows at `L1` and `L2`. The same result could not be obtained using numeric abstraction alone since the crucial invariant at `L0` is a disjunction of numeric terms. Intuitively, this means that executions on which `cond` is true and executions on which it is false must be considered separately. Of course, the combination is not more powerful than predicate abstraction, but it is much more efficient, since a single predicate `cond` (along with other numeric terms) is sufficient to express the desired invariant.

Table 2: Summary of implementations of NUMPREDDOM

Name	Value	Example	Num.
NEXPOINT	$2^{2^P} \times N$	$(p \vee q) \wedge (0 \leq x \leq 5)$	EXP
NEX	$2^P \mapsto N$	$(p \wedge 0 \leq x \leq 3) \vee (q \wedge 1 \leq x \leq 5)$	EXP
MTNDD	$2^P \mapsto N$	$(p \wedge 0 \leq x \leq 3) \vee (q \wedge 1 \leq x \leq 5)$	SYM
NDD	$2^P \mapsto 2^N$	$(p \wedge (x = 0 \vee x = 3) \vee (q \wedge (x = 1 \vee x = 5)))$	SYM

(a)

```
int cond;
char p[5],q[8],*x;

x = cond ? p : q;
if(cond)
  while(x < p + 5)
    *x++ = 'a';
else
  while(x < q + 8)
    *x++ = 'a';
```

(b)

```
p.offset = 0; p.alloc = 5;
q.offset = 0; q.alloc = 8;
if(cond) {
  x.base = p.base; x.offset = p.offset;
  x.alloc = p.alloc;
} else {
  x.base = q.base; x.offset = q.offset;
  x.alloc = q.alloc;
}
L0: if(cond) {
  while(FPDATA(x) < FPDATA(p) + 5) {
L1: assert(x.offset < x.alloc);
    *FPDATA(x) = 'a';
    x.offset = x.offset + 1;
  }
} else {
  while(FPDATA(x) < FPDATA(q) + 8) {
L2: assert(x.offset < x.alloc);
    *FPDATA(x) = 'a';
    x.offset = x.offset + 1;
  }
}
```

Figure 9: A Code Fragment (a) and Its Instrumentation (b)

4.6 Abstraction Refinement

Predicate abstraction is effective only when combined with a technique to infer the appropriate set of predicates. This is achieved via a process called counter-example guided abstraction refinement (CEGAR). In general, CEGAR is an area of active research [Henzinger 2004, Gulavani 2008]. The problem of refining a combination of numeric and predicate abstractions is the subject of our ongoing work. In our current PANA implementation,

we use the following naive refinement strategy. Given an execution trace CE leading to a potential buffer overflow (a counterexample), we extract a set of constraints, known as an UNSAT core. Specifically, the UNSAT core is a syntactic subformula of the weakest preconditon of CE that is also unsatisfiable. Intuitively, the UNSAT core explains why the counterexample is infeasible. If the core is empty, the counterexample is feasible and no refinement is needed. Otherwise, we add all of the numeric variables appearing in any constraint in the UNSAT core to the numeric part of the combined abstract domain. If these variables are already part of the domain, we add the constraints appearing in the UNSAT core to the predicate part of the domain.

Recall that the INSTRUMENTATION phase of COVERT reduces buffer manipulations to numeric operations over the buffers' attributes, for example, `offset`, `base`, etc. In addition, a precise analysis must know the values of specific predicates at specific program locations, for example, the value of arguments to `assert`. Therefore, successful analysis of programs generated via COVERT's INSTRUMENTATION requires keeping track of the values of both numeric variables and predicates, and the way they influence each other. We believe that this makes PANA particularly suited as the analysis engine of COVERT due to its combination of numeric abstraction and predicate abstraction. Our belief is vindicated by our experimental validation, described in Chapter 5.

5 Implementation and Validation

The INSTRUMENTATION phase is implemented on top of the CIL infrastructure for instrumentation of C programs [UC Berkeley 2010]. By default, CIL simplifies the C program, thereby achieving Step 1 of INSTRUMENTATION [Necula 2002]. The rest of the steps are implemented using the extensive mechanisms for rewriting C programs provided by CIL. In our implementation, the bodies of instrumentation functions, such as `fp_strcpy_alarm` and `fp_strcat_alarm`, are inlined at their call sites; variants like `fp_strcpy_alarm` and `fp_strcpy_no_alarm` are merged into a single function.

The ANALYSIS phase was implemented using BLAST, COPPER, and PANA. BLAST and COPPER are written in Ocaml and C++ respectively, and use SIMPLIFY for theorem proving. PANA is written in JAVA, and uses CVCLITE for theorem proving, APRON library for numeric abstraction, and CUDD for manipulating Binary Decision Diagrams. Theorem proving is used for constructing predicate abstraction, for deciding feasibility of abstract counterexamples, and for constructing UNSAT cores for refinement. For our experiments with PANA, we use the NEX combination from Table 2.

Table 3: BLAST, COPPER, and PANA Comparison

	`sendmail`								`wu-ftpd`			
	Safe				Unsafe				Unsafe			
	Total	Crash	Time	Speedup	Total	Crash	Time	Speedup	Total	Crash	Time	Speedup
BLAST	32	32	0.0	*	24	10	66.0	2.2	11	0	56.6	4.5
COPPER	32	11	254.7	4.0	24	8	350.4	6.2	11	0	16.9	1.3
PANA	32	0	235.8	1.0	24	0	84.9	1.0	11	0	12.6	1.0

5.1 Experimental Validation

We evaluated COVERT using a suite of benchmarks created by Zitser and colleagues from vulnerable and safe versions of open source programs `sendmail` and `wu-ftpd` [Zitser 2004]. When Zitser and colleagues analyzed these examples with static analysis tools, they yielded many false warnings, low error detection rates, and poor disambiguation between safe (i.e., with no overflows) and unsafe (i.e., with overflows) versions. Thus, these examples are known to be hard for static analysis. From this suite, we constructed 67 different test cases that were analyzed successfully by PANA with no false warnings. The average test case consisted of about 2000 lines of C. Out of these 67 test cases, 32 and 35 are safe and unsafe, respectively. Table 3 summarizes our results. In the table, Total = total # of test cases; Crash = # of crashes; Time = Total running time (in seconds) for cases that the tool completed successfully; Speedup = Factor by which total running time of PANA on the successful cases of this tool is smaller. A "Crash" means that BLAST or COPPER either crashed or terminated with an incorrect result. BLAST crashes in 42 cases, while COPPER crashes in 19 cases. Note that the successful test cases for BLAST and COPPER overlap but are not exactly the same.

The chart in Figure 10 shows a complete breakdown of running times for all three tools. A negative value indicates a crash. We see that PANA appears to be superior to both BLAST and COPPER in terms of analyzing the test cases successfully. Specifically, in 56 out of 67 cases, PANA outperforms both COPPER and BLAST, sometimes by over a factor of 20 in terms of running time. In the remaining 11 cases, PANA is slower, but the running times are less than 1.5 seconds, and hence negligible. Overall, the running times for PANA have the least variance. Our results indicate that COVERT is successful in analyzing real programs for buffer overflows with no false warnings. Moreover, the CHECKER based on PANA is superior to those based on BLAST and COPPER.

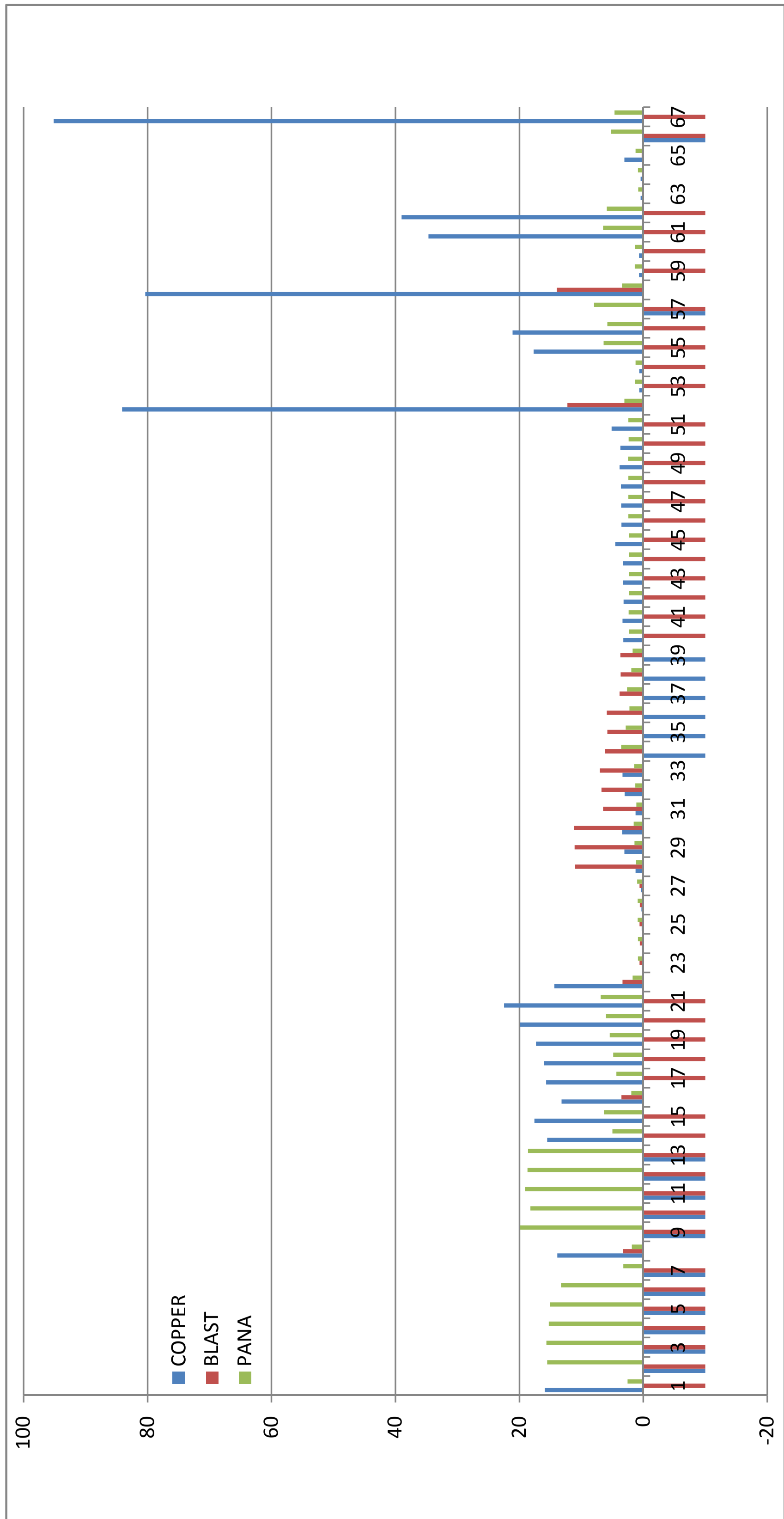

Figure 10: Bar Chart Showing Comparison of Running Times Between COPPER*,* BLAST*, and* PANA *as the* CHECKER *for All Tests*

6 Conclusion

Buffer overflows continue to be a major source of software vulnerabilities. Despite recent advancements, finding overflows statically and effectively remains an open challenge. Automated software verification is an active area of investigation, and improved tools and techniques continue to emerge from the research and development efforts in this domain. Therefore, an application framework that is able to leverage the power of the latest software verification tools for finding buffer overflows would be extremely useful. This report presents such a framework, called COVERT.

Broadly, COVERT works in two phases: INSTRUMENTATION and ANALYSIS. During INSTRUMENTATION, the target C program is instrumented such that buffer overflows are reduced to assertion violations. In the ANALYSIS phase, a static software verification tool, called CHECKER, is used to check for assertion violations in the instrumented code, and to generate error reports. We implemented and evaluated COVERT on a set of benchmarks derived from real programs. For the ANALYSIS phase, we experimented with three instances of CHECKER—BLAST, COPPER and PANA. Our results indicate that COVERT is effective at reducing the number of false warnings, while remaining scalable.

Our results with COVERT are encouraging, but preliminary. Ideally, we envision that COVERT would be integrated within a full-fledged software development environment, and used routinely by programmers, and quality-control engineers. In fact, an ultimate deployment of COVERT would be used like a compiler, and it would emit warnings, errors, and other appropriate diagnostic feedback related to buffer overflows. Such a deployment would also be able to use an array of state-of-the-art verification tools in a seamless manner. Many of the remaining issues requiring resolution to get to this point are essentially those of robust software development, and we believe that wider industrial support is needed to achieve this goal.

References

URLs are valid as of the publication date of this document.

[Ball 2001] Ball, T. & Rajamani, S. K. "Automatically Validating Temporal Safety Properties of Interfaces", 103–122. Dwyer, M. B., editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN '01)*, volume 2057 of *Lecture Notes in Computer Science*. Toronto, Canada, May 19–20, 2001. New York, NY: Springer-Verlag, May 2001.

[Beyer 2005] Beyer, D.; Henzinger, T. A.; Jhala, R.; & Majumdar, R. "Checking Memory Safety with BLAST", 2–18. Cerioli, M., editor, *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE '05)*, volume 3442 of *Lecture Notes in Computer Science*. Edinburgh, UK: Springer-Verlag, April 2005.

[CERT CC 2001] CERT CC. "*Code Red Worm Exploiting Buffer Overflow in IIS Indexing Service*", 2001. http://www.cert.org/advisories/CA-2001-19.html.

[Chaki 2005] Chaki, S.; Ivers, J.; Sharygina, N.; & Wallnau, K. "The ComFoRT Reasoning Framework", 164–169. Etessami, K. & Rajamani, S. K., editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV '05)*, volume 3576 of *Lecture Notes in Computer Science*. Edinburgh, Scotland, July 6–10, 2005. New York, NY: Springer-Verlag, July 2005.

[Clarke 2003] Clarke, E. M.; Grumberg, O.; Jha, S.; Lu, Y.; & Veith, H. "Counterexample-Guided Abstraction Refinement for Symbolic Model Checking". *Journal of the ACM (JACM) 50*, 5 (September 2003): 752–794.

[Clarke 2004] Clarke, E.; Kroening, D.; & Lerda, F. "A Tool for Checking ANSI-C Programs", 168–176. Jensen, K. & Podelski, A., editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, volume 2988 of *Lecture Notes in Computer Science*. Barcelona, Spain, March 29–April 2, 2004. New York, NY: Springer-Verlag, March–April 2004.

[Cousot 1977] Cousot, P. & Cousot, R. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints", 238–252. *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '77)*. Los Angeles: Association for Computing Machinery, January 1977.

[Cowan 2000] Cowan, C.; Wagle, P.; Pu, C.; Beattie, S.; & Walpole, J. "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade", 119–129. *Proceedings of the DARPA Information Survivability Conference and Expo (DISCEX)*. Hilton Head, South Carolina, January 25–27, 2000. Los Alamitos, CA: IEEE Computer Society, January 2000.

[Dahn 2003] Dahn, C. & Mancoridis, S. "Using Program Transformation to Secure C Programs Against Buffer Overflows", 323–333. van Deursen, A.; Stroulia, E.; & Storey, M.-A. D., editors, *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE '03)*. Victoria, BC, Canada, November 13–16, 2003. Los Alamitos, CA: IEEE Computer Society, 2003.

[Dhurjati 2006] Dhurjati, D. & Adve, V. S. “Backwards-Compatible Array Bounds Checking for C with Very Low Overhead”, 162–171. Osterweil, L. J.; Rombach, H. D.; & Soffa, M. L., editors, *Proceedings of the 28th International Conference on Software Engineering (ICSE ’06)*. Shanghai, China, May 20–28, 2006. New York, NY: Association for Computing Machinery, May 2006.

[Dor 2003] Dor, N.; Rodeh, M.; & Sagiv, S. “CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C”, 155–167. *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI ’03)*. San Diego, CA, June 9–11, 2003. New York, NY: Association for Computing Machinery, June 2003.

[Ganapathy 2003] Ganapathy, V.; Jha, S.; Chandler, D.; Melski, D.; & Vitek, D. “Buffer Overrun Detection Using Linear Programming and Static Analysis”, 345–354. Jajodia, S.; Atluri, V.; & Jaeger, T., editors, *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS ’03)*. Washington, D.C., October 27–30, 2003. New York, NY: Association for Computing Machinery, October 2003.

[Graf 1997] Graf, S. & Saïdi, H. “Construction of Abstract State Graphs with PVS”, 72–83. Grumberg, O., editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV ’97)*, volume 1254 of *Lecture Notes in Computer Science*. Haifa, Israel, June 22–25, 1997. New York, NY: Springer-Verlag, June 1997.

[Gulavani 2008] Gulavani, B. S.; Chakraborty, S.; Nori, A. V.; & Rajamani, S. K. “Automatically Refining Abstract Interpretations”, 443–458. Ramakrishnan, C. R. & Rehof, J., editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS ’08)*, volume 4963 of *Lecture Notes in Computer Science*. Budapest, Hungary: Springer-Verlag, March–April 2008.

[Gurfinkel 2008] Gurfinkel, A. & Chaki, S. “Combining Predicate and Numeric Abstraction for Software Model Checking”, 127–135. *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD ’08)*. Portland, OR: IEEE Computer Society, November 2008.

[Henzinger 2002] Henzinger, T. A.; Jhala, R.; Majumdar, R.; & Sutre, G. “Lazy Abstraction”, 58–70. *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’02)*, volume 37(1) of *SIGPLAN Notices*. Portland, OR, January 16–18, 2002. New York, NY: Association for Computing Machinery, January 2002.

[Henzinger 2004] Henzinger, T. A.; Jhala, R.; Majumdar, R.; & McMillan, K. L. “Abstractions From Proofs”, 232–244. Jones, N. D. & Leroy, X., editors, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’04)*. Venice, Italy: Association for Computing Machinery, January 2004.

[Hovemeyer 2005] Hovemeyer, D.; Spacco, J.; & Pugh, W. “Evaluating and Tuning a Static Analysis to Find Null Pointer Bugs”, 13–19. *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE ’05)*. Lisbon, Portugal: Association for Computing Machinery, September 2005.

[Jarzombek 2004] Jarzombek, J. "Systems, Networks and Information Integration Context for Software Assurance". http://www.sei.cmu.edu/acquisition/start/publications/asconferenceonsis2004.cfm, January 2004.

[Jones 1997] Jones, R. & Kelly, P. "Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs", 13–26. Kamkar, M., editor, *Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG '97)*, volume 2(009) of *Linkoping Electronic Articles in Computer and Information Science*. Linkoping, Sweden, May 26–27, 1997. Linkoping, Sweden: Linkoping University Electronic Press, May 1997.

[Miné 2006] Miné, A. "The Octagon Abstract Domain". *Higher-Order and Symbolic Computation 19*, 1 (March 2006): 31–100.

[MITRE-CWE-09 2009] "2009 CWE/SANS Top 25 Most Dangerous Programming Errors", 2009. http://cwe.mitre.org/top25.

[Necula 2002] Necula, G. C.; McPeak, S.; Rahul, S. P.; & Weimer, W. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs", 213–228. Horspool, R. N., editor, *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*, volume 2304 of *Lecture Notes in Computer Science*. Grenoble, France, April 8–12, 2002. New York, NY: Springer-Verlag, April 2002.

[Necula 2005] Necula, G. C.; Condit, J.; Harren, M.; McPeak, S.; & Weimer, W. "Ccured: type-safe retrofitting of legacy software". *ACM Transactions on Programming Languages and System (TOPLAS) 27*, 3 (May 2005): 477–526.

[NIST 2002] NIST. *The Economic Impacts of Inadequate Infrastructure for Software Testing* (02-3). : National Institute of Standards and Technology (NIST), 2002. National Institute of Standards and Technology (NIST) Planning Report 02-3, http://www.nist.gov/director/planning/upload/report02-3.pdf.

[Ruwase 2004] Ruwase, O. & Lam, M. S. "A Practical Dynamic Buffer Overflow Detector", 159–169. *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04)*. San Diego, CA, February 5–6, 2004. Reston, VA: Internet Society, February 2004.

[Schmidt 1998] Schmidt, D. A. "Data Flow Analysis is Model Checking of Abstract Interpretations", 38–48. *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*. San Diego, CA, January 19–21, 1998. New York, NY: Association for Computing Machinery, January 1998.

[Shankar 2001] Shankar, U.; Talwar, K.; Foster, J. S.; & Wagner, D. "Detecting Format String Vulnerabilities with Type Qualifiers", 201–216. *Proceedings of the 10th USENIX Security Symposium*. Washington, D.C., August 13–17, 2001. Berkeley, CA, August 2001.

[UC Berkeley 2010] UC Berkeley. "*CIL Infrastructure for C Program Analysis and Transformation (v 1.3.7)*". http://www.eecs.berkeley.edu/∼necula/cil/, 2010.

[Wagner 2000] Wagner, D.; Foster, J. S.; Brewer, E. A.; & Aiken, A. "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities". *Proceedings of the 7th Annual Network and Distributed System Security Symposium (NDSS '00).* San Diego, CA, October 31–November 5, 2004. Reston, VA: Internet Society, 2000.

[Zitser 2004] Zitser, M.; Lippmann, R.; & Leek, T. "Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code", 97–106. *Proceedings of the 12th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE '04).* Newport Beach, CA, October 31–November 5, 2004. New York, NY: Association for Computing Machinery, October–November 2004.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	August 2010	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
COVERT: A Framework for Finding Buffer Overflows in C Programs via Software Verification	FA8721-05-C-0003

6. **AUTHOR(S)**

Sagar Chaki, Arie Gurfinkle

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2010-TR-X209

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-2010--029

11. **SUPPLEMENTARY NOTES**

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. **ABSTRACT (MAXIMUM 200 WORDS)**

Buffer overflows continue to be the source of a vast majority of software vulnerabilities. Solutions based on runtime checks incur performance overhead, and are inappropriate for safety-critical and mission-critical systems requiring static—that is, prior to deployment—guarantees. Thus, finding overflows statically and effectively remains an important challenge. This report presents COVERT, an automated framework aimed at finding buffer overflows in C programs using state-of-the-art software verification tools and techniques. Broadly, COVERT works in two phases: INSTRUMENTATION and ANALYSIS. The INSTRUMENTATION phase is the core phase of COVERT. During INSTRUMENTATION, the target C program is instrumented such that buffer overflows are transformed to assertion violations. In the ANALYSIS phase, a static software verification tool is used to check for assertion violations in the instrumented code, and to generate error reports. COVERT was implemented and then evaluated it on a set of benchmarks derived from real programs. For the ANALYSIS phase, experiments were conducted with three software verification tools – BLAST, COPPER, and PANA. Results indicate that the COVERT frame-work is effective at reducing the number of false warnings, while remaining scalable.

14. SUBJECT TERMS	15. NUMBER OF PAGES
buffer overflows, software verification, CHECKER, PANA, instrumentation	35

16. **PRICE CODE**

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL