Buffer overflow analyzer

Jonathan Lam & Daniel Tsarev

December 18, 2021

Contents

1 Abstract

Buffer overflows are a very well-known form of software vulnerability in which memory buffer overruns allow attacker to read and/or write memory locations, in some cases allowing them to inject crash or take control of a system. They are prevalent in low-level software written in memory-unsafe languages, such as and especially web servers.

We aim to create a simple static analysis tool that tracks the use of stack-allocated buffers to known memory-unsafe functions in C source code, such as the notorious `strcpy` standard library function. This is achieved using a dataflow analysis and call graph tracing on the unoptimized LLVM IR generated using the clang compiler frontend. We develop a novel dataflow analysis method, called "buffer origin dataflow analysis," that allows us to track assignments of buffers. We evaluate this method and describe its shortcomings. A sample implementation can be found on GitHub at @jlam55555/overflow-analyzer.

2 Motivation

The motivation for this project was the first lab[1], where the source code of a toy web server was analyzed and buffer overflows were used to gain unauthorized access. After doing the lab by hand, we wondered if similar vulnerabilities could be detected automatically via static analysis, and were motivated to create a tool that does so.

2.1 Threat model

There are a few closely-related buffer-overflow attacks. Most involve the overflowing of fixed-size local buffers, which are stored on the stack. The stack also contains important information about program control flow that should not be accessible by the programmer or a user of the program. Some low-level languages like C are memory-unsafe and do not check bounds or pointer accesses, and thus overflowing buffers may overwrite these critical locations in memory. A malicious attacker can craft special messages including their own arbitrary code ("shellcode") that are intended to be placed in these buffers, such that they hijack the control flow of the program. This is especially problematic if the application is running as a privileged user.

A seminal paper in the literature of buffer overflows is the paper "Smashing the stack for fun and profit" by Aleph One. Over time, there have been numerous attempts at curbing stack overflows, such as stack canaries, address space layout randomization (ASLR), no-execute (NX) stack memory pages, memory-safe languages, better error warnings about unsafe usages. However, as attackers get more creative (e.g., "return-to-libc" attacks to bypass the NX-bit or heap-spraying to bypass ASLR), there is no complete solution for a C/C++ or assembly programmer working at the low-level OS or firmware level.

If we consider the STRIDE security model, almost all facets are compromised due to the low-level nature of the code. The main concerns are:

Tampering The integrity of the compromised program is no longer secure, along with any data that the program interacts with.

Denial of service Buffer overflows may lead to a program crash.

Elevation of privilege As low-level services tend to be running with special users with special privileges, an attacker with control of the program now has access to these privileges.

Since buffer overflows typically attack a low-level system or service, the entire service becomes compromised. In the case of complex services with many user interactions, such as a webserver, this may compromise the security with spoofing, information disclosure, and repudiability at the application level.

[1]MIT 6.858 Lab 1: Buffer overflows

2.2 Risk analysis

While buffer overflow attacks are one of the oldest and best known attacks, the OWASP Foundation classifies the severity as “very high” and the likelihood of exploit as “high to very high,” and states that “buffer overflow attacks against both legacy and newly-deveilop applications are still quite common”[2].

A quick search for buffer overflow vulnerabilities on the Common Vulnerabilities and Exposures (CVE) lists 12323 exploits, 765 vulnerabilities in 2021 alone[3]. Buffer overflow vulnerabilities may cause data leakage or downtime, which are very expensive. Gartner, a technology consulting company, estimates that the typical company loses \$5,600 per minute of downtime[4]; IBM estimates that the average data leak for a U.S company costs \$4.24 million in 2021[5]. Needless to say, buffer overflow attacks are very prevalent, and the damage is costly.

[2] `https://owasp.org/www-community/vulnerabilities/Buffer_Overflow`

[3] `https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer+overflow`, counted by number of search results at time of writing

[4] `https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/`

[5] `https://www.ibm.com/security/data-breach`

3 Methods

Buffer overflow analysis is performed through a dataflow analysis on a LLVM IR representation of the source code input. Explanations of IRs, LLVM, and dataflow analysis will be explained in the following sections.

3.1 Review of LLVM and intermediate representations

LLVM is a set of modular libraries used to aid in compiler implementation. One of it core features is a low-level, consistent, human-readable and easily programmable **intermediate representation** (IR). This IR can be thought of as a modern, high-level, machine-independent assembly language. A number of compiler front-ends (i.e., the component of a compiler that manages parsing a program to an abstract syntax tree (AST)) target LLVM, including the C compiler clang. Our implementation uses clang to generate LLVM code from a C source file.

Some general compilers terminology may be helpful when referring to the LLVM IR:

Instruction An atomic statement that models assembly instructions.

Virtual register (VR) An abstract register that is used for storing temporary values from instructions; can be thought of as a variable in the IR.

Basic block A sequence of instructions that do not involve branching (control flow).

Control flow graph (CFG) A sequence of basic blocks that models the flow of control of a program.

Instruction set architecture (ISA) A set of instructions that an abstract model of a CPU can execute.

Three-address IR/ISA A form of IR where each instruction has an operation code (opcode), up to two parameter virtual registers, and an optional destination virtual register.

Load/store IR/ISA An IR/ISA that tends to adhere to the simpler-instruction model of RISC ISAs. In particular, memory operations (such as `load`/`store`) are explicit and separated from computation.

LLVM is a three-instruction, load-store IR. A program is contained in a `llvm::Module`, which stores a collection of global variables and `llvm::Functions`; each `llvm::Function` is a collection of `llvm::BasicBlocks`; and each `llvm::BasicBlock` is a collection of `llvm::Instructions`.

LLVM also includes the concept of a VR, called `llvm::Value`, which has a type (`llvm::Type`). There is a dual between VRs and instructions; each instruction in a three-address ISA has a target VR (if it has a target; otherwise the target value is of type `void`). As a result, each instruction is identified by its target value, and thus every instruction *is a* value (i.e., `llvm::Instruction` subtypes `llvm::Value`). It may seem strange in the implementation to be using instructions as values, but this is due to this equivalence between instructions and VRs.

3.1.1 Difficulties in OCaml

Our original goal was to implement our analysis program in OCaml, since it is a program often used in programming languages (PL) theory and has a port of LLVM. However, we found that the OCaml LLVM package is outdated, so we switched to the native implementation language, C++.

3.2 Review of dataflow analysis

Dataflow analysis is a principled approach to statically tracking some property throughout the source code of a program, and forms the basis of many static analysis approaches. The type of property, i.e. the thing that we want to analyze, is highly dependent on the type of dataflow analysis.

We may consider **zero analysis** as an example for minimum understanding. In zero analysis, we track whether a variable is zero at any point in the program. We list the instructions in a basic block vertically, and list the variables we want to analyze horizontally, so that we have a table, as shown in Table 1. We start with an initial analysis of all unreachable ($\perp$) abstract values. After each instruction, we update the analysis based on the instruction and the previous analysis, so that the invariant (property) is preserved – the rule that determines the next analysis as a function of the previous analysis and the current instruction is called the transition function.

In the case of zero analysis, we have four abstract values that may determine the possible analyses of a variable:

$$L = \{\perp, Z, N, \top\}$$

where Z means definitely zero, N means definitely non-zero, $\top$ is a special value that means indeterminate (Z or N). $\perp$ is a special value that means that this value has not been analyzed yet, or unreachable, and is useful for simplifying the analysis.

Zero analysis is defined by the following abstraction function α which defines the invariant that we are tracking. In this case, if a variable evaluates to zero, then it is given the abstract value Z; if it evaluates to some non-zero value, then it is given the abstract value N. (Note that there are also the additional values $\perp$ and $\top$ when neither of these analysis hold true.)

$$\alpha(v) = \begin{cases} Z & \text{if } v \Downarrow 0 \\ N & \text{if } v \Downarrow n \wedge n \neq 0 \end{cases}$$

The transition function describes the analysis transition after an instruction. It is defined piecewise on the instruction type. Assume x, y are variables to

Instruction	x	y	z
	$\bot$	$\bot$	$\bot$
$x \leftarrow 0$	Z	$\bot$	$\bot$
$y \leftarrow 1$	Z	N	$\bot$
$z \leftarrow y$	Z	N	N
$x \leftarrow z + x$	Z	N	N
$x \leftarrow y - z$	$\top$	N	N

Table 1: Sample zero analysis

analyze, and c is a constant, and our language is composed only of assignments, additions, and subtractions.

$$f(x \leftarrow c, \sigma) = \begin{cases} \sigma[x \mapsto Z], & \text{if } c = 0 \\ \sigma[x \mapsto N], & \text{else} \end{cases}$$

$$f(x \leftarrow y, \sigma) = \sigma[x \mapsto \sigma_y]$$

$$f(z \leftarrow x \pm c, \sigma) = \begin{cases} \sigma[z \mapsto \sigma_x], & \text{if } c = 0 \\ \sigma[z \mapsto N], & \text{if } \sigma_x = Z \wedge c \neq 0 \\ \sigma[z \mapsto \top], & \text{else} \end{cases}$$

$$f(z \leftarrow x \pm y, \sigma) = \begin{cases} \sigma[z \mapsto \sigma_x], & \text{if } \sigma_y = Z \\ \sigma[z \mapsto \sigma_y], & \text{if } \sigma_x = Z \\ \sigma[z \mapsto \top], & \text{else} \end{cases}$$

Hopefully this simple example gives enough intuition into dataflow analysis. More detailed explanations of some of the terms are given below.

Program point A point in the program's runtime, i.e., a conceptual state if the program were paused between two sequential instructions. A dataflow analysis is defined for each program point, and is identified by the instruction I that precedes it.

Tracked variable This represents a physical element of the program to analyze. For example, we can perform the analysis on each variable in the source code. Tracked variables (or simply "variables") are denoted v.

Abstract value A member of a lattice that represents some information about the dataflow analysis. This represents some property of the tracked variables that we want to track for this analysis. An abstract value is denoted l, and the set (a lattice) that it is defined on is denoted L.

Abstraction function A function $\alpha(v) = \sigma_v$ mapping an tracked variable to an abstract value. This is the invariant that must be maintained by the transition function.

Analysis information (Referring to an instruction, basic block, or function): All of the tracked variables and their corresponding abstract values in the given instruction/basic block/function. Sometimes abbreviated to simply "analysis." Denoted using σ. For the abstract value associated with a single variable v, we can denote this as σ_v.

Lattice An algebraic structure used to represent a partial ordering on a potentially infinite set. In dataflow analysis, we define the subsumption partial order $\sqsubseteq$ on the set of abstract values L. In this context, we pronounce the expression $L_1 \sqsubseteq L_2$ as "L_2 is more general than L_1". We can informally extend this to an analysis on all tracked variables of an analysis, i.e., $\sigma_1 \sqsubseteq \sigma_2 \iff \forall v \in \sigma_1 \cap \sigma_2 . \sigma_{1_v} \sqsubseteq \sigma_{2_v}$. A valid transition function must produce a more general analysis than its input, i.e., $\sigma \sqsubseteq f(I, \sigma)$. For the purposes of our analysis, every lattice is a meet- and join-semilattice, which include the upper-bound $\top$ (pronounced "top") and the lower-bound $\bot$ (pronounced "bottom").

Transition function A function $f(I, \sigma_v)$ taking a program point and an analysis, and returning the updated analysis after the instruction such that $\alpha(v) = f(I, \sigma_v)$; in other words, we can never "narrow" the analysis, but only "widen" it with additional information that generalizes over all known states, especially in the case of combining multiple incoming edges (joining analyses). As before, this can be extended to the entire analysis σ rather than a single variable.

Join operation A binary function $l_1 \wedge l_2$ on a lattice that produces a minimum upper-bound to l_1 and l_2 (unique given a join-semilattice). As before, this can be naturally extended to an analysis $\sigma_1 \wedge \sigma_2$. The join operation is used to "join" the outgoing dataflow analyses of all incoming edges to a basic block, which then forms the initial dataflow analysis for that basic block.

The dataflow analysis is performed by invoking the transition function at each program point and joining incoming edges of basic blocks until the program reaches a fixpoint (i.e., until no operations change the analysis of the output program point). This is performed on a per-function basis using a simple worklist algorithm, described later; the per-function analyses are then used by the interprocedural analysis, which is described in the next section.

We defer to Chapter 4 of the *Program Analysis* textbook (linked under References) for an introduction to dataflow analysis.

3.3 Review of interprocedural analysis

We can extend dataflow analysis to multiple functions in multiple ways. One way is to simply inline all functions and perform a regular dataflow analysis; however, this method is costly and impossible for (mutually) recursive functions. This forces us to not inline function calls, but treat the analysis as a series

```c
void f(char *d, char *s) {
    strcpy(d, s);
}

int main() {
    char d1[512], s1[512], d2[1024], d1[1024];
    f(d1, s1);
    f(d2, s2);
}
```

Figure 1: Issue with specificity of global dataflow analysis

of links, as a call graph. (Note that this is more complicated than a CFG, since no branches occur in the middle of a basic block; a function may invoke another function at any point within the caller, so we need to keep the analysis information at every program point of the caller).

There is some difficulty if we perform the interprocedural analysis in the same manner as the dataflow analysis, i.e., if we converge to a fixed point globally (over all functions). Consider the example shown in Figure 1. If a global fixpoint were achieved, then the analysis σ_d of the parameter `d` would include both `d1` and `d2`, and the analysis σ_s would be `s1` and `s2`. This analysis is not incorrect; these are all the possible values that `d` and `s` can hold over the course of the program. However, we might assume that there is an unsafe `strcpy` due to this, even when this is not the case. The problem is a matter of **specificity** – we assume that all invocations of `f` share the same context and analysis, and this is a general issue with dataflow analysis – it is conservative.

As a result, we perform **context-sensitive** analysis, in which a function's analysis is independent of other invocations of the functions by the caller. However, since this may be non-terminating (in the case of recursion), we limit the analysis until we reach a fixed point in the functions on the call stack; i.e., the analysis of a function will generalize any previous analysis of the function from mutually recursive calls, if any[6].

We simplify the analysis by assuming that the transition function for any function call is the identity function[7] – i.e., functions do not affect the analysis in the current function. Without this simplification, the cached analysis changes not only as a function of the parameters but also as a function of all of its contained function calls (who themselves are functions of their parameter set and internal function calls, and so on). This has two implications: functions cannot modify the analyses (i.e., reassign) of their parameters that are buffers, and functions cannot return a buffer type. Luckily, these two operations are not too common in cases that would affect our analysis (e.g., returned pointers are usually pointing to heap-allocated buffers).

[6]We are not sure if this is conventional.

[7]We are also not sure if this is conventional.

```
char buf[512], **pbuf, *buf2;
pbuf = &((char*)buf);
buf2 = *pbuf;
```

Figure 2: Importance of storing pointers to buffer origins

We defer to Chapters 8 and 10 of the *Program Analysis* textbook (linked under References) for more details on interprocedural analysis, such as a proof of precision and termination.

3.4 Buffer origin dataflow analysis (BODA)

3.4.1 Overview

Buffer A fixed-size stack-allocated (local variable) array.

Candidate buffer A virtual register that has an array or pointer type.

Buffer origin (Of a candidate buffer) Refers to the set of possible buffers or parameter candidate buffers that a candidate buffer may be *referring to*.

Note that when discussing buffer origins, “referring to” is only in the sense of memory equality (referential semantics), not in terms of the equality of its contents (value semantics). For example, after performing `strcpy(dst,src)`, `dst`’s analysis is not updated.

There are multiple ways to hold a reference to a value. For example, we want to be able to handle the scenario shown in Figure 2. In this example, we want `buf2` to refer to `buf`. We also know that `pbuf` refers to `buf`, but it is not equal to `buf` – instead, we maintain in our analysis that it may be a reference to `buf`, denoted as `&buf2`. Similarly, we want to keep track of all virtual registers that may refer to a buffer via more levels of indirection, i.e., `&&buf2`, `&&&buf2`, etc. This also works out nicely given that LLVM uses a load/store instruction set, in which we often have pointers referring to buffers. For example, variable definitions are implicitly pointers to the value type: the variable declaration `char buf[]` defines the virtual register `%buf` with type `[i8]*`.

3.4.2 Definition

The abstraction function for the BODA analysis is shown in (1). For a given candidate buffer b, the BODA analysis returns the set of all possible buffer origins may be referred to by b.

$$\alpha(b) = \{\text{possible buffer origins of } b\} \tag{1}$$

In this case, our lattice structure is the set lattice. For any set lattice, $\bot = \varnothing$, $\top =$ the set of all buffer origins, and the join operator ($\wedge$) is set union.

The transition functions for BODA analysis are defined below. (2) describes the analysis for an array declaration. (3), (4), and (5) address all memory operations (assignment) in LLVM. LLVM's `getelementptr` is more complex than a simple `load`, treating it as we treat the `load` opcode simplifies our analysis. The final rule, (6), is the catchall; for any non-assignment operator, the transition function ignores the instruction.

$$f(\texttt{\%buf = alloca [i8]*}, \sigma) = \sigma[\texttt{\%buf} \mapsto \{\texttt{\&buf}\}] \tag{2}$$

$$f(\texttt{store i8* \%p1, i8** \%p2}, \sigma) = \sigma[\texttt{\%p2} \mapsto \{\&b_o : b_o \in \sigma_{\texttt{\%p1}}\}] \tag{3}$$

$$f(\texttt{\%p2 = load i8** \%p1}, \sigma) = \sigma[\texttt{\%p2} \mapsto \{*b_o : b_o \in \sigma_{\texttt{\%p1}}\}] \tag{4}$$

$$f(\texttt{\%p2 = getelementptr i8** \%p1, 0, 0}, \sigma) = \sigma[\texttt{\%p2} \mapsto \{*b_o : b_o \in \sigma_{\texttt{\%p1}}\}] \tag{5}$$

$$f(\texttt{\%a = }\boxed{\texttt{op}}\texttt{ \%b, \%c}, \sigma) = \sigma \tag{6}$$

3.4.3 Assumptions and caveats

Some major assumptions of the dataflow analysis note are listed below.

Limited to fixed-size stack-allocated buffers Only fixed-size buffers are tracked as possible buffer origins. Pointer variables are not regarded as possible buffer origins. Pointers allocated via other methods (heap allocation, or variable-length arrays (VLAs)) cannot be tracked due to requiring runtime information. Some cases of this may be trackable with the aid of constant propagation, but that would unnecessarily complicate our implementation.

No pointer arithmetic In an instruction such as `p=b+3`, where `p` is a pointer and `b` is a buffer, `b` is considered a buffer origin of `p`, even if it is not exactly the case. Future work may attempt to account for this.

Function calls do not affect dataflow analysis This means that the transition function for all function call instructions is the identity function. This has two implications: functions cannot modify the analysis of their parameters (e.g., a function cannot swap the buffer origins of its parameters), and functions cannot return a buffer. Allowing for this would greatly complicate the interprocedural analysis, and would require research to solve efficiently. We feel that omitting this should not detract much from the analysis, since functions do not often modify what their input buffers point to, and functions do not usually return pointers to stack-allocated buffers (returned buffers tend to be heap-allocated).

```
function BodaWorklist($f$)
    $B \leftarrow$ basic blocks in $f$
    mark $B_0$ dirty
    $B_d \leftarrow B[0]$
    while $B_d$ is not empty do
        $b \leftarrow$ pop from $B_d$
        if $b$ is dirty then
            analyze($b$)
            mark $b$ clean
            if $b$ not at fixpoint then
                for all $b_s \in B : b_s$ succeeds $b$ do
                    mark $b_s$ dirty
                    insert $b_s$ into $B_d$
                end for
            end if
        end if
    end while
end function
```

Figure 3: The worklist algorithm for basic blocks in a function

3.4.4 Per-function analysis: buffer origin dataflow analysis

The implementation of the dataflow analysis is fairly standard for a dataflow analysis, following the worklist algorithm for computing a fixpoint (at the function level) and using the join operator and transition function to update the analysis (at the basic-block/instruction level). See Figure 3 for the pseudocode of the worklist algorithm. This is a generic formulation that can be applied to any dataflow analysis.

The specifics of the choice of basic block to pop may affect the efficiency of the worklist algorithm (see Kildall's algorithm) but that is not the goal here and was not implemented. In our implementation, B_d is implemented as a simple LIFO queue.

3.4.5 Interprocedural analysis: CFG tracing (using BODA data)

See Figure 4 for the pseudocode for tracing the call graph. At each function invocation, each argument buffer is replaced with its potential buffer origins. This process happens recursively.

The algorithm as stated doesn't handle (mutual) recursion, which will cause an infinite loop. One method to achieve convergence (other than limiting recursion depth) is to use a previous analysis of the function on the call stack as the initial value for an analysis. If there is no change in the analysis of a particular function invocation, then the recursion stops. In other words, there is a convergence among the analysis of the functions along the call stack. We did not finish this in our implementation.

```
function TraceCallGraphRec($f, O$)
    for all function invocations $i_g() \in f$ do
        $p_g \leftarrow [O_{i_g}/O]p_g$
        if $g$ is dangerous then
            warn about dangerous usage
        else if $g$ is user-defined then
            TraceCallGraphRec($g, p_g$)
        end if
    end for
end function

function TraceCallGraph($M$)
    $f_m \leftarrow$ main function of $M$
    TraceCallGraphRec($f_m, \{\}$)
end function
```

Figure 4: Trace call graph algorithm (not accounting for mutual recursion)

4 Samples and results

Several sample programs and their generated outputs are shown below. The outputs shown below are from the current implementation with the `-DDEBUG` compile flag turned off; with the flag on, much more information is shown about the dataflow analysis. We omit this for brevity.

Figure 5 shows the simplest example of buffer tracing: a pointer referencing a local buffer. In this case, the source buffer in `strcpy` will always be shorter than the destination buffer, which is safe if `b3` contains a well-formed (null-terminated) string. However, there is always the chance that this is not the case, so we warn about `strcpy` anyways.

Figure 6 shows an example where a pointer's referenced buffer is indeterminate. Since there is a possible path such that the source buffer is longer than the destination buffer, we warn about the dangerous usage.

Figure 7 shows an example of interprocedural analysis. Through the function call, the parameters are replaced with their buffer origins when analyzing the function call. The call to `strcpy` is able to determine that it uses an unsafe combination of source and destination buffers located in the `main` function.

Figure 8 shows another example of interprocedural analysis, demonstrating the recursive nature of the call graph tracing. We can track dangerous buffer usages from arbitrarily deep in the call stack.

Figure 9 illustrates the problem with the context-insensitivity of dataflow analysis noted in Figure 1. We are able to consider different function invocations as separate contexts, and thus avoid the false positive that a global dataflow analysis would flag.

Other noteworthy examples are loops and recursion. Loops (even non-terminating ones) should converge in the analysis. Analyzing (mutual) recursion does not terminate in our current implementation. We would also like to include more debugging information, such as backtrace of dangerous usages and source file position (line and column) of the declarations of buffer origins, in addition to the function name.

```c
int main() {
        char b1[8192], *b2, b3[512];
        b2 = b1;
        strcpy(b2, b3);
}
```

(a) Source

```
[Warning]: unsafe function strcpy()
        Possible destinations: main:b1[8192]
        Possible sources: main:b3[512]
```

(b) Output

Figure 5: Sample program: simple buffer tracking

```c
int main() {
        char src1[8192], src2[512], *src, dst[512];
        if (rand() % 2) {
                src = src1;
        } else {
                src = src2;
        }
        strcpy(dst, src);
}
```

(a) Source

```
[Warning]: unsafe function strcpy()
        Possible destinations: main:dst[512]
        Possible sources: main:src1[8192] main:src2[512]
        [Danger]: potential buffer overflow (source:8192 > destination:512)
```

(b) Output

Figure 6: Sample program: indeterminate buffer origin

```
void f(char *dst, char *src) {
        strcpy(dst, src);
}

int main() {
        char b1[512], b2[4096];
        f(b1, b2);
}
```

(a) Source

```
[Warning]: unsafe function strcpy()
        Possible destinations: main:b1[512]
        Possible sources: main:b2[4096]
        [Danger]: potential buffer overflow (source:4096 > destination:512)
```

(b) Output

Figure 7: Sample program: buffer tracking through function invocations

```
void g(char *s) {
        char buf[512];
        strcpy(buf, s);
}

void f(char *s) {
        g(s);
}

int main() {
        char str[8192];
        f(str);
}
```

(a) Source

```
[Warning]: unsafe function strcpy()
        Possible destinations: g:buf[512]
        Possible sources: main:str[8192]
        [Danger]: potential buffer overflow (source:8192 > destination:512)
```

(b) Output

Figure 8: Sample program: multi-level buffer tracking

```
void f(char *d, char *s) {
        strcpy(d, s);
}

int main() {
        char b1[512], b2[512], b3[1024], b4[1024];
        f(b1, b2);
        f(b3, b4);
}
```

(a) Source

```
[Warning]: unsafe function strcpy()
        Possible destinations: main:b1[512]
        Possible sources: main:b2[512]
[Warning]: unsafe function strcpy()
        Possible destinations: main:b3[1024]
        Possible sources: main:b4[1024]
```

(b) Output

Figure 9: Sample program: specificity of call graph tracing

5 Conclusion

Buffer overflows are a very prevalent software vulnerability in low-level programming. We attempt to address common cases of buffer overflows by deep tracing of stack-allocated buffers throughout functions to well-known dangerous functions, using a custom dataflow analysis called buffer origin analysis. Our analysis successfully covers the given test scenarios, and is conservative in its analysis (as dataflow analysis tends to be). To achieve greater accuracy, call-graph tracing is applied for function calls.

This method of dataflow analysis, while applied in BODA to buffers, can be applied to track any local variable pointers throughout a program. We are not aware of any use cases other than buffer overflow detection at this time.

Due to a number of strong assumptions made in the analysis, there are a large number of uncovered scenarios that can be addressed in the future, in order to address more advanced use cases. Our algorithm may also be improved by handling recursive cases using the call-stack convergence method described earlier, as well as by improving logging output to use LLVM debugging info. It will also be informative to examine our program on more complicated programs than our artificial test cases, and to examine the implemenatation of existing static analysis tools and how they handle buffer overflow vulnerabilities.

6 Resources

***Program Analysis* (Textbook)** `https://www.cs.cmu.edu/~aldrich/courses/17-396/program-analysis.pdf`