Automatic Heap Exploit Generation

Brent Lim Tze Hao, Advised by Professor David Brumley
Carnegie Mellon University, Pittsburgh, PA
{*brentlim, dbrumley*}*@cmu.edu*

Abstract

The automatic exploit generation (AEG) challenge is, given a program, automatically find vulnerabilities and generate exploits for them. Avgerinos et al showed that, given the source code of the program, AEG was possible for certain stack smashing and format string exploits. In Automatic Heap Exploit Generation (AHEG), we do away with the need for source code and we extend AEG to automatically find heap bugs and generate heap exploits on applications running on Windows XP SP3. Our contributions are: 1) we develop Simple ASM, a simpler subset of x86 ASM, on which we develop our algorithms on, 2) we propose memory tagging, a technique used to infer the *class* of data, as opposed to the *type* of data, base on semantic analysis of the program, 3) we introduce "2-steps exploits", which extends Avgerino's approach to exploit generation to heap exploits 4) we build an end-to-end system that takes executables on Windows XP SP3 and automatically generates crashing inputs against them.

1 Introduction

Avgerinos et al showed that it is possible to automatically discover vulnerabilities and generate control flow exploits, given only the source code of the target program [1]. The main idea of his work was to use preconditioned symbolic execution to quickly search for pathological paths in the program and imposing additional constraints on the crashing input so that the solution to these constraints is the exploit string.

This work builds heavily on and extends the work by Avgerinos et al. In particular, this work addresses one weakness of AEG: AEG was limited to stack-based buffer overflows and format string exploits because it did not have semantic information about user bytes in memory. For example, it could not explicitly assert the constraint on user input that the user input be of a certain length.

This paper develops techniques and a proof-of-concept for automatic heap exploit generation (AHEG) on executables compiled to run on Windows XP SP 3 on an x86 architecture and stripped of debugging information.

Contributions

1. We develop Simple ASM, a subset of the x86 instruction set rich enough to express real-world programs. We provide the translation from x86 to Simple ASM, and show techniques developed in AEG extend to programs expressed in Simple ASM.
2. We propose *memory tagging*, a technique used to infer the *class* of data, as opposed to the *type* of data, based on semantic analysis of the program. For example, consider `malloc( strlen( get_user_input() ) )` and `malloc(get_user_input())`. In both examples, the user can control the size of memory allocated by `malloc` and in both cases, the function `malloc` expects an argument of *type* **int**. However, the actual user input to *influence* the same outcome differs. In other words, *the predicate imposed on the user input differs*. For example, to get the program to

allocate 5 bytes of memory, in the first case, we might provide the user input "aaaaa" and in the second case, we might provide the number "5". Memory tagging is important because in assembly, it is not immediately obvious that the input to a function, in this case `malloc`, comes directly from user input or as a result of operations that are semantically equivalent to `strlen`.

3. We introduce "2-steps exploits", an explicit construction of the predicates on the input space to generate exploits against doubly-linked linked list, commonly used in the implementations of heap allocators, including the Windows Heap Manager. Brumley et al showed that exploit generation can be automated by characterizing exploits as predicates on the program state space [2], hence the limitations of AEG to stack-smashing attacks and format string attacks can be overcome by constructing predicates for different classes of exploits.
4. We build an end-to-end system that takes executables running on Windows XP SP3 and automatically generate crashing inputs as proof-of-concept of the techniques introduced in this paper.

2 Background

2.1 Bug Find

In AHEG, we are interested in finding *exploitable bugs*, which are flaws in programs that allow an attacker to perform arbitrary code execution. Formally, we are searching for paths in programs which lead to violations of enforceable security polices [18], in particular, that EIP does not contain user input.

Three popular techniques employed today in software verification to search for such pathological paths are taint analysis [8] [15], symbolic execution [12] [7] [4] [5] [3] and concolic execution [20].

Taint Analysis The idea of taint analysis is to identify certain sources from which user input, also known as *tainted* input, is introduced, such as from sockets, files or command line, and to propagate the *taint* according to a *taint policy*. This technique is a form of data flow analysis and is used to identify the bytes in memory which user has direct control over. A useful application of this technique is to check that EIP is never tainted.

Forward Symbolic Execution Instead of supplying the program with real, or *concrete*, user input, in forward symbolic execution, we emulate the program with *symbolic* bytes. Each time we condition branch on a *symbolic* byte, we fork a new interpreter and explore both branches simultaneously. We also impose contraints on the symbolic bytes at each branch, so that a string satisfying these constraints will take the same path in the program. For more information on taint analysis and forward symbolic execution, we refer the reader to a survey by Schwartz et al [19].

Concolic execution Concolic execution is a variant of symbolic execution, except that instead of emulating the program with *symbolic* user input, we run the program with *concrete* user input. We then instrument the program and treat these *concrete* bytes as *symbolic*, so that whenever we condition branch on user input, we impose an additional constraint, which we can negate and solve for to get another input that traverses a different path in the program. The main advantage concolic execution has over symbolic execution is that in implementing concolic execution, we do not have to keep track of the program state for every branch.

AHEG employs a combination of all three techniques.

2.2 Exploit Generation

In APEG [2] and AEG [1], exploit generation is reduced to the problem of generating predicates on the input space so that the exploit is the satisfying input to the predicates. In AHEG, we take this approach further by explicitly constructing predicates for different classes of exploits. The key insight in constructing such predicates is in encoding the *influence* the user has on specific bytes in memory. The most obvious *influence*

is *direct influence*, in which the byte in memory is exactly what the user entered. Another *influence* is *transformation influence*, in which the byte in memory has been transformed by a series of operations (such as ADD, SUB, etc) from the original user input, so that to control the byte in memory, we have to apply an inverse to the series of operations to get the required user input. This inverse is usually performed by a solver. Both of these *influence* can be identified via data flow analysis and are used in APEG and AEG. However, there exists other *influence* over bytes which user might have indirect control over. To use the same example in the introduction, in *malloc(strlen(user_input))*, the user can control the argument of *malloc* by varying the size of the input buffer. By extending symbolic execution to loops, Saxena et al showed that we can identify such *length influence* [17].

3 Overview

In this section, we show how the different components and algorithms in `AHEG` fit together at a high level. We elaborate on each component in later sections.

3.1 Motivating example

Consider the program shown in Figure 1. The program first reads a number from an input file. Then it copies that many bytes of user input into the buffer called `buf`. The program has only 2 end-states, one that outputs "Very good", and one that outputs "not very good". To get to the former state, the number and the length of user input must match the magic number. In the rest of this section, we will show how `AHEG` arrives at this user input. This example mimics the parsing of file formats, which is a common source of heap vulnerabilities in programs. [1]

3.2 Initialization

Machine code to Simple ASM All analysis and algorithms described in this paper are performed on programs written in Simple ASM (§ 4). `AHEG` performs the translation from x86 Intel assembly instructions to Simple ASM dynamically. The example program is first compiled to an x86 binary, which is then instrumented and executed natively. For each native instruction the program executes, the instrumentor translates the instruction into the corresponding instruction(s) in Simple ASM.

Seed input `AHEG` employs concolic execution to explore the state space of the program. Thus, we need to provide `AHEG` with a seed input with which to run the example program against. Figure 2 shows the seed input that we have provided.

3.3 Starting the concolic execution

We initialize the global Path Constraints (*PC*) to true. Each time we encounter a `CondJump` instruction, and we take the true branch, we update $PC := PC \wedge c$, where c is the condition of the conditional jump. Otherwise, we take the false branch and we update $PC := PC \wedge \neg c$. In addition, we perform the following analysis:

Syntactic analysis As the name suggests, in syntactic analysis, we look at instructions individually, and do not maintain context information of instructions executed. For example, in *Maximum Buffer Heuristic* (§ 6.1), we identify instructions that subtracts a constant from the stack pointer, and take the maximum of all these constants. In the example program, the maximum was 1308, from the instruction `SUB esp, $0x51c`, which came from the function `setSBUpLow`, which is part of the Windows C-Runtime (CRT) start up code that was added to the example program when we compiled it.

[1] For example, ClamAV had a heap bug in the code to parse PE (Portable Executable) files [10]. Another example is the multiple vulnerabilities in Adobe Acrobat and Adobe Reader due to improper parsing of JBIG2-encoded data in PDF files (CVE-2009-0509 to CVE-2009-0512 and CVE-2009-0888, CVE-2009-0889)

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int magic_number = 42;

  char buf[255];
  char line1[255];
  char line2[255];

  int len = 0;
  int i = 0;

  FILE *fp;

  fp = fopen("input", "r");

  if (fp == NULL) {
    return 3;
  }

  if (fgets(line1, 255, fp) == NULL) {
    return 1;
  }

  if (fgets(line2, 255, fp) == NULL) {
    return 2;
  }

  len = atoi(line1);

  if (len < 12) {
    return;
  }

  strncpy(buf, line2, len);

  if (strlen(buf) == magic_number) {
    printf("Very good");
  }
  else {
    printf("not very good");
  }

  fclose(fp);

  return 0;
}
```


Figure 1: Example 1 - This program has only 2 end-states. "Very good" and "Not very good". To get to the first state, length of buf and len must match the magic number.

13
aaaaaaaaaaaaa

Figure 2: Seed input to start off concolic execution on example program

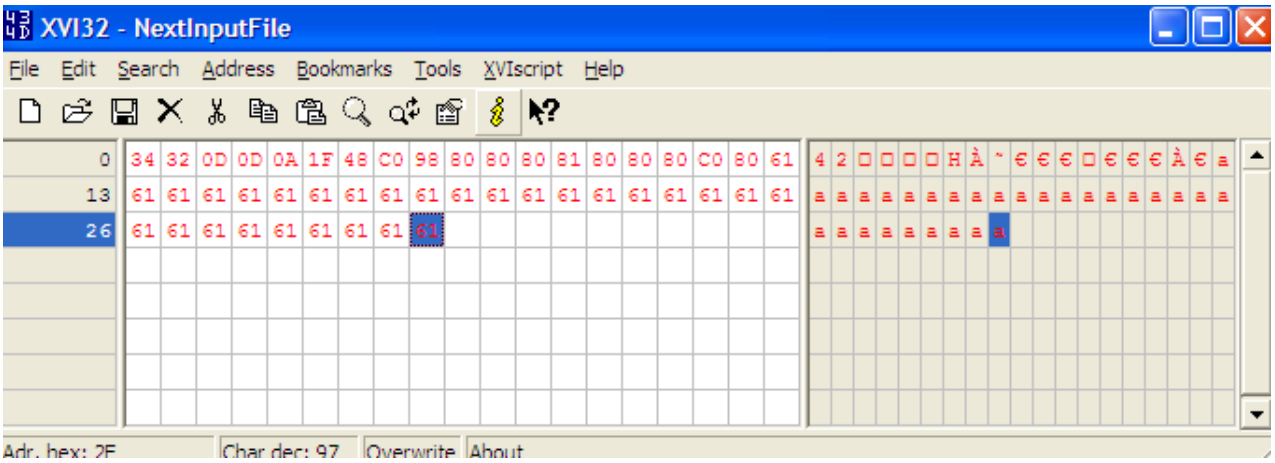

Figure 3: Output when `AHEG` was run with program in Figure 1

Semantic analysis We look at instructions in the context of other instructions executed, which as a group, perform operations that are semantically equivalent to operations we define. In particular, we are looking for blocks of code that perform the equivalent of one of the following operations: (1) Unbounded Copying, (2) Bounded copying (3) `strlen` (§ 5.3). Semantic analysis reveals that although `strncpy` was used, since the bound depended on user input, namely `len`, the function was really an Unbounded Copy and that the size of `buf` depended on the size of `line2` and `len`. In particular, `length(buf)` = min(`length(line2)`, `len`).

Memory tagging Memory tagging (§ 5.1) occurs at the instruction level and at the semantic block level. At the instruction level, `line1` and `line2` are marked as containing user input, because they originate from user input. `atoi(line1)` is a series of operations performed on line1, and due to *tag propagation*, `len` is marked as containing user input. More importantly, `len` is tagged in such a way that we can imagine `len` = f(line1), so that to set `len` to any value, say 42, we need only apply the inverse of f, denoted f^{-1}, to get the required input, i.e. `line1` = $f^{-1}(42)$. At the semantic block level, we tag `buf` as a buffer that comes from the buffer at `line2`. We also tag `strlen(buf)` as the value that is equal to `length(buf)`.

3.4 Generating the next input

After the first iteration of concolic execution with the seed input provided (Figure 2), the final path constraints are $PC = (\texttt{len} \geq 13) \wedge (\texttt{length(buf)} = \min(\texttt{length(line2)}, \texttt{len})) \wedge (\texttt{length(buf)} \neq 42)$). To get the next input, we negate the last clause in the path constraint, to get $PC' = (\texttt{len} \geq 13) \wedge (\texttt{length(buf)} = \min(\texttt{length(line2)}, \texttt{len})) \wedge (\texttt{length(buf)} = 42))$ Solving for this constraints, we get `len` = 42 and `length(buf)` = 42. From f^{-1}, we know we have to set `line1` to the ascii values $0x340x32$, for "4" and "2" respectively. From Unbounded Copy, we know that `buf` was the destination of an unbounded copy from `line2`, and hence `length(line2)` has to be 42. To achieve that, `AHEG` generates 42 random characters, for the final output shown in Figure 3.

```
<sasm-ins> ::= <DOP-ins> | <JMP-ins> | <UserInput-ins>
<cond> ::= EQ | NE | GE | GT | LE | LT
<reg> ::= EAX | EBX | ECX | EDX | ESI | EDI | EBP | ESP
<mem> ::= 0 | 1 | 2 | 3 | ... | 255
<operand> ::= <reg> | (<reg>) | [(<reg>)] | [<mem>] | $<mem>
<BOP> ::= ADD | SUB | MULT | DIV | OR | AND | XOR
<DOP> ::= MOV | <BOP>
<DOP-ins> ::= <DOP> <operand> <operand>
<JMP-ins> ::= UncondJump <operand> | CondJump <operand> <operand> <cond
    > <operand>
<UserInput-ins> ::= ReadFile(<operand>, <operand>, <operand>) |
    ReadArgv(<operand>, <operand>, <operand>)
```

Figure 4: Definition of Simple ASM instruction

4 Simple ASM

We simplify the usual 32-bit x86 architecture: in Simple ASM, we have byte-addressable memory model with 2^8 entries and a processor with 8 8-bit general purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP), and an instruction register EIP. Instructions are stored on memory together with data. Note that although Simple ASM appears to be assembly language for an 8-bit architecture, our implementation of `AHEG` simulates a 32-bit architecture by providing 32 8-bit registers, EAX1, EAX2, EAX3, EAX4, EBX1, etc and grouping registers into their corresponding registers, e.g. EAX = (EAX1, EAX2, EAX3, EAX4). Simple ASM is defined so that a register holds 1 byte of data to simplify analysis.

Figure 4 shows the definition of Simple ASM instructions ⟨`sasm-ins`⟩. At the high level, instructions in Simple ASM are divided into 3 categories:

Data flow instructions These are the binary operations, `MOV`, `ADD`, `SUB`, `MULT`, `DIV`, `OR`, `XOR` and `AND`, which take *destination* and *source* operands (in that order), perform the corresponding operation and save the results in the *destination* operand. An operand is a general purpose register or an immediate value. If the destination operand is an immediate value, the results are not saved. This is to emulate instructions like `CMP` and `TEST`, which are essentially `SUB` and `AND` instructions respectively, except that the results of the operations are not saved. Data flow instructions propagate user input, or *taint*, as commonly known in taint analysis.

Control flow instructions These are `UncondJump`, which takes one operand and sets EIP to the value of the operand, and `CondJump`, which takes four operands, namely 2 operands to compare, the *cond* operand and the *location* operand. `CondJump` checks the *cond* operand against the 2 operands we are comparing, and if the conditions are met, sets EIP to the value of the *location* operand. Otherwise, it sets EIP to the address of the next instruction. Control flow instructions propagate control flow information, or *implicit flow*.

User input source These are the functions which introduces user input into the system. In SimpleASM, there are only 2 sources of user input, SymFile and SymArgv. Both `ReadFile` and `ReadArgv` take 3 arguments, *buf*, *NumBytesToRead*, and *&NumBytesRead*, where *buf* is the memory address of the start of the buffer where user input is copied into, *NumBytesToRead* is the maximum number of user input to copy, and *& NumBytesRead* is the memory address to store the actual number of bytes copied.

Macros Given an operand x, (x) refers to the value of x. If x is a register, then (x) is the value stored in the register, which is always an immediate value. Otherwise, x is an immediate value and $(x) = x$. Given an integer y, $[y]$ refers to the value stored in memory at the address y and $\$y$ refers to the immediate value y.

4.1 Translating from x86 Assembly to SimpleASM

The instruction set in SimpleASM is a strict subset of x86 assembly, i.e., there are certain instructions in x86 assembly that are not expressible in SimpleASM. For example, there is no notion of a system call (the INT instruction) in SimpleASM (apart from the 3 that are built into SimpleASM). However, for all other instructions that depend only on the general purpose registers and memory (as opposed to the CR registers, or floating point registers, for example), they are expressible in SimpleASM. The idea is that in SimpleASM, each instruction either read/write from/to register/memory, so we can express instructions that depends on register and memory as one or more instructions in SimpleASM.

5 Memory Tagging

Definition Let A be a set of user input. Let f be a function from a set of user input A to the set $\mathbb{N} \cup \{\perp\}$. Then the *influence* the user has over a byte i, is a function f_i, such that if $f_i(x) = n, x \in A, n \in \mathbb{N}$, the value of byte i when the program terminates is n. The value of the byte is *undetermined* if $f_i(x) = \perp$. The user has no *influence* over byte i if $f_i(x) = \perp, \forall x \in A$.

Suppose we are given a program P, and we have found a path in the program which performs *Uncond-Jump* (i), where i is some address in memory. Given f_i, we can then ask the question, does there exist $x \in A$, such that $f_i(x) = n$, for some n we choose? If the answer is yes, we have essentially found an input that controls EIP, i.e. a control-flow hijack exploit. We call x the *satisfying input* such that $f_i(x) = n$, and given f_i, we can recover x with the help of constraint solvers. Hence, the goal is to find f_i.

This section describes *memory tagging*, a technique used to approximate f_i. The technique itself involves a number of algorithms, namely *Buffer Inference* (§ 5.4) and loop-extended symbolic execution (§ 5.2). Loop-extended symbolic execution was first introduced by Saxena et al [17]. Both algorithms rely on the accurate detection of loops, which is described in § 5.2. In addition, *Buffer Inference* also require, as input, blocks of code which are semantically equivalent to Unbounded Copy, Bounded Copy and `strlen`. We identify such blocks of code in *Semantic Analysis*, described in § 5.3.

The key idea of *memory tagging* is it encodes the *influence* the user has over a byte and approximates f_i. For each byte in memory, we are interested in the amount of control we have of that byte as a function of user input. § 5.1 introduces the 4 tags, DAT, SYM, VAR and PTR, used in *Memory Tagging*

5.1 Tag descriptions

Definition A *tag* is a piece of data attached to each byte which provides semantic information about the byte. We write tag_i to mean the tag attached to the byte at address i. We also tag bytes in registers. We write $tag_{reg}, reg \in \{EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP\}$ for the tag attached to the byte in register *regname*.

In AHEG, we introduce 5 tags, DAT, SYM, VAR, BUF and PTR. We provide a high level overview of each tag in this subsection. There are two ways tags can be introduced and propagated: 1) at the instruction-level and 2) at the *block* level. A *block* is a group of instructions, that as a whole, perform the semantic equivalent of a pre-defined function. For example, Figure 9 shows a block that performs a `strlen` of a buffer. Table 1 and 2 summarizes the tag introduction and propagation rules at the instruction level and block level respectively. Block-level tagging is done at the last instruction in the block.

Instruction	Rule
`ReadFile(buf, n, &m)`	Suppose `ReadFile(buf, n, &m)` reads k bytes of user input from the symbolic file. Then (1) $tag_{buf+i} :=$ DAT$(i,$ `SymFile`$)$, $\forall 0 \leq i < k$. (2) Let var be the variable id of a fresh symbolic variable. $tag_m := VAR(var)$. (3) $PC := PC \wedge (var = length(0,$ `SymFile`$)) \wedge (var < n)$
`ReadArgv(buf, n, &m)`	Suppose `ReadArgv(buf, n, &m)` reads k bytes of user input from the symbolic file. Then (1) $tag_{buf+i} :=$ DAT$(i,$ `SymArgv`$)$, $\forall 0 \leq i < k$. (2) Let var be the variable id of a fresh symbolic variable. $tag_m := VAR(var)$. (3) $PC := PC \wedge (var = length(0,$ `SymArgv`$)) \wedge (var < n)$
MOV [mem], $c	$tag_{mem} := \emptyset$
MOV reg, $c	$tag_{reg} := \emptyset$
MOV [mem1], [mem2]	$tag_{mem1} := tag_{mem2}$
MOV [mem], reg	$tag_{mem} := tag_{reg}$
MOV reg, [mem]	$tag_{reg} := tag_{mem}$
MOV reg1, reg2	$tag_{reg1} := tag_{reg2}$
BOP [mem], $c	If $tag_{mem} = \emptyset$, then $tag_{mem} := \emptyset$. Otherwise, $tag_{mem} := SYM(BOP(tag_{mem}, \$c))$
BOP reg, $c	If $tag_{reg} = \emptyset$ then $tag_{reg} := \emptyset$. Otherwise $tag_{reg} := SYM(BOP(tag_{reg}, \$c))$
BOP [mem1], [mem2]	If $tag_{mem2} = \emptyset$ then use the rule *BOP* [mem], $c, where mem = mem1 and c = [mem2]. Otherwise, $tag_{mem2} \neq \emptyset$. If $tag_{mem1} = \emptyset, tag_{mem1} := SYM(BOP(\$[mem1], tag_{mem2}))$. Otherwise, $tag_{mem1} := SYM(BOP(tag_{mem1}, tag_{mem2}))$
BOP [mem], reg	If $tag_{reg} = \emptyset$ then use the rule *BOP* [mem], $c, where c = (reg). Otherwise, $tag_{reg} \neq \emptyset$. If $tag_{mem} = \emptyset, tag_{mem} := SYM(BOP(\$[mem], tag_{reg}))$ Otherwise, $tag_{mem} := SYM(BOP(tag_{mem}, tag_{reg}))$
BOP reg, [mem]	If $tag_{mem} = \emptyset$ then use the rule *BOP* reg, $c, where c = [mem]. Otherwise, $tag_{mem} \neq \emptyset$. If $tag_{reg} = \emptyset, tag_{reg} := SYM(BOP(\$(reg), tag_{mem}))$. Otherwise $tag_{reg} := SYM(BOP(tag_{reg}, tag_{mem}))$
BOP reg1, reg2	If $tag_{reg2} = \emptyset$ then use the rule *BOP* reg, $c, where c = (reg2). Otherwise $tag_{reg2} \neq \emptyset$. If $tag_{reg1} = \emptyset, tag_{reg1} := SYM(BOP(\$(reg1), tag_{reg2}))$ Otherwise, $tag_{reg1} := SYM(BOP(tag_{reg1}, tag_{reg2}))$.
CondJump op1, op2, relation, branch	If branch is taken, $PC := PC \wedge (tag_{op1}$ relation $tag_{op2})$. Otherwise $PC := PC \wedge \neg(tag_{op1}$ relation $tag_{op2})$ If branch is taken and DAT $\in tag_{branch}$, then output PC as potential exploit
UncondJump branch	If DAT $\in tag_{branch}$, output PC as potential exploit

Table 1: List of tag introduction/propagation rules associated with each Simple ASM instruction. Recall that *BOP* $\in$ {ADD, MULT, SUB, DIV, OR, AND, XOR} and *relation* $\in$ {EQ, NE, GE, GT, LE, LT}

Semantic block	Rules
Unbounded Copy $(I, dest, src, tc)$	If $tag_{src} = BUF(id, n)$, then (1) new Buffer(*buffer_id* := dest, *actualLength* := getBuffer(id).actualLength - n, *potentialLength* := getBuffer(id).potentialLength - n, *srcBuffer* := id, *srcBufferOffset* := n) (2) $tag_{dest+i} := BUF(dest, i), \forall 0 \leq i < getBuffer(dest).actualLength$
Bounded Copy $(I, dest, src, bound, tc)$	If $tag_{src} = BUF(id, n)$, then (1) new Buffer(*buffer_id* := dest, *actualLength* := getBuffer(id).actualLength - n, *potentialLength* := min(bound, getBuffer(id).potentialLength - n), *srcBuffer* := id, *srcBufferOffset* := n) (2) $tag_{dest+i} := BUF(dest, i), \forall 0 \leq i < getBuffer(dest).actualLength$
strlen (tc, buf)	$PC := PC \wedge (tc = length(buf))$

Table 2: List of tag introduction/propagation rules associated with each semantic block

DAT tag If a byte has been tagged DAT(x, y), where x is a number and $y \in \{SymArgv, SymFile\}$, then that byte is under *direct influence* from the x^{th} byte of user input from the command line or from a file. This means that to set that byte to a particular value, say *42*, we need only set the x^{th} byte of user input to *42*.

Recall that in Simple ASM, there are only two functions that retrieve user input, namely *ReadArgv* and *ReadFile*. In both cases, we have a buffer containing user input and we tag each byte in the buffer with *DAT* since these buffers came directly from user input.

SYM tag If a byte has been tagged SYM(y), where y is an expression involving a set A of user input bytes, then the byte is under *transformation influence* from the user input bytes $x_1, ..., x_n \in A$. This means that to set the byte to a particular value, say z, we need to set $x_1, ..., x_n$, so that they evaluate to z in y.

Whenever either operands of a data flow instruction, except MOV, is tagged, we might introduce a SYM tag to the destination operand. The exact rules governing the introduction of SYM tags are listed Table 1, but intuitively, the SYM tag "remembers" all the operations performed on user input. For example, given the instruction ADD EAX, EBX, where [eax] = 5, and EBX is tagged with DAT(7, SymArgv), then we will tag EAX with SYM(ADD(5, DAT(7, SymArgv)). If later, we encounter yet another instruction, say SUB ECX, EAX, where ECX is tagged with DAT(5, SymArgv), then we retag ECX with the tag SYM(SUB(DAT(5, SymArgv), SYM(ADD(5, DAT(7, SymArgv)))), which means that ECX really is the difference between the 7^{th} and 5^{th} byte of user input and the sum of the constant 5.

PTR tag The tag PTR(n, p), where n is a unique *buffer id* (§ 5.4), and p, an address in memory, tells us that the value of this byte has been used as a pointer to access address p, i.e. this byte has been *dereferenced*. Since in `AHEG`, *buffer ids* are exactly the address of the buffers, we shorten pointers tags to one argument, namely PTR(n).

VAR tag The tag VAR(var) represents a symbolic variable, with variable id *var*.

BUF tag If a byte has been tagged BUF(id, n), then the byte is the n^{th} byte of buffer with buffer id *id*.

Multiple tags It is possible that a byte can be tagged in more than one way. For example, if a byte that user controls is used to dereference memory, then it will be tagged with DAT and PTR. In these cases, we concatenate the tags DAT . PTR.

5.2 Loop detection

Definition In AHEG, we define a *loop* to be a block of code such that, each piece of instruction in the block, not necessarily contiguous, is executed at least once, and whenever the block is done executing, a

```
1. MOV EAX, 4;
2. ADD EAX, $1;
3. CondJump EAX, $3, GE, 8;
4. UncondJump (EAX);
5. UncondJump 1;
6. UncondJump 1;
7. UncondJump 1;
```

Figure 5: The blocks are {1, 2, 3, 4, 5}, {1, 2, 3, 4, 6} and {1, 2, 3, 4, 7}. The fundamental reason why we are unable to detect this form of loop lies in instruction 4 - the address of the instruction that we execute next depends on a value computed within the loop, which we are trying to detect in the first place.

certain condition is tested to determine if the same block should be re-executed. The test is part of the block. Hence, the only way to exit from a loop is via a `CondJump`. This is not the most general definition of a loop (Figure 5 shows a block of code that is intuitively a loop, but not defined to be a loop), but accurate detection of the most general loops is beyond the scope of this project.

`AHEG` adopts the loop detection algorithm presented in LoopProf [14], which is capable of detecting loops dynamically in the absence of static information like the control flow graph of the program. `AHEG` slightly modifies the algorithm to account for *exit conditions* of loops.

Detecting exit conditions Intuitively *exit conditions* is a set of conditions, such that if any of the condition in the set is true, then we exit the loop. Algorithm 1 outputs a set P that is a subset of P' of exit conditions. It is a subset because there might be certain exit conditions we do not detect as we are doing dynamic analysis and there might be paths we do not explore that could potentially be exit paths out of the loop.

At a high level, the algorithm maintains 2 sets, C and P, where C is the constraints governing the current path and P is the set of exit conditions. Whenever we encounter a `CondJump`, we first check the invariant that at least one of the branches lies in the loop. This is because in our definition of a loop, we require every instruction to have been executed at least once. Hence for us to encounter a `CondJump`, the subsequent instruction must belong to either the true or false branch and must also be encountered. Hence at least one of the branches must lie in the loop. Then, every time we find a path that exits the loop, we update P with the constraints of the path. Otherwise, if we remain in the loop after a `CondJump`, we update C with the additional constraint imposed by that jump.

Loop-extended Symbolic Execution Loop extended symbolic execution was introduced by Saxena et al [17] to capture the relationship between data that would otherwise be marked concrete in traditional symbolic execution and user data. An example is shown in Figure 7. The variable i does not depend on user data directly, but instead depends on the number of iterations of the while loop, which the user can control. We adopt the same idea of introducing symbolic *trip counts* (tc) per loop, but we modify their algorithm that detects for loop-dependent variables, to better integrate with the existing framework that we have built.

Instead of performing symbolic program analysis to link loops to input, we use a REP tag to symbolically represent that a specific operation on a byte should be repeated n many times, where n is a symbolic argument to REP. Given a set I of instructions in a loop, we generate a set C of bytes that are affected by at least one instruction in the loop. Now, for each $i \in C$, let x_i be the memory tag of byte i before we enter the loop and let $h_i(x_i)$ be the memory tag of i after the first iteration of the loop. Recall that Memory Tagging allows us to "remember" all the operations performed on a byte. Hence, subsequent iterations of the loop will result

in the memory tag $h_i(h_i(...h_i(x)...))$. To symbolically reason about iterations, we introduce the tag REP, and tag x_i with $REP(x_i, h_i, tc)$ to mean perform h_I on x_i tc many times, where tc is the symbolic trip-count of the loop.

Algorithm 1: Algorithm to identify set P, subset of set of exit conditions of a loop

input : I, the set of instructions in a loop
output: P, the set of exit conditions

```
1  C ← true;
2  P ← ∅ ;
3  ins ← GetCurrentIns() ;
4  while ins ∈ I do
5      if ins = CondJump then
6          c ← GetCondition(ins) ;
7          nextIns ← GetNextIns() ;
8          tb ← GetTrueBranch(ins) ;
9          tf ← GetFalseBranch(ins) ;
10         if tb ∉ I and tf ∉ I then
               /* This can never happen, because in our definition of
                  loop, we require each instruction to have been executed
                  at least once.  If tb ∉ I and tf ∉ I, then it must be that
                  ins ∉ I, violating the condition of this while loop     */
11             Error() ;
12         if tb ∉ I then
13             P ← P ∪ {C ∧ c} ;
14         if tf ∉ I then
15             P ← P ∪ {C ∧ ¬c} ;
16         if nextIns = tb then
17             C ← C ∧ c ;
18         if nextIns = tf then
19             C ← C ∧ ¬c ;
20     ins ← nextIns ;
21 return P ;
```

5.3 Semantic analysis

We identify blocks of code that performs the semantic equivalent of (1) Unbounded copying, (2) Bounded copying and (3) `strlen`. Table 3 summarizes the output of each analysis.

Unbounded copying First, we define the template $T_{\texttt{copy}} = (\langle A := mem[src], src++, mem[dest] := A, dest++\rangle, \{A, src, dest\}, \emptyset)$ and employ the semantics-aware matching algorithm described by Christodorescu et al [6] to identify the block of code within the program that does the semantic equivalent of a buffer copy. If the block of code sits inside a loop, as detected by a loop-detection analysis, and if the *exit condition* of the loop is the predicate $mem[src] = c$ for some constant c (example $c = '\setminus 0'$ the delimiting NULL byte in strings in C) and the user has *influence* over $mem[src]$, then the loop is semantically equivalent to an unbounded copy. Intuitively, if the loop only checks the source buffer for a byte that the user controls to determine if the loop

Semantic block	Output	Description
Unbounded Copy	$(I, dest, src, tc)$	I - a set of instructions that together, perform the Unbounded Copy operation. *dest* - address of the first byte of the destination buffer. *src* - address of the first byte of the source buffer. tc - is a symbolic variable representing the number of times the instructions in I are executed
Bounded Copy	$(I, dest, src, bound, tc)$	I - a set of instructions that together, perform the Bounded Copy operation. *dest* - address of the first byte of the destination buffer. *src* - address of the first byte of the source buffer. *bound* - an integer that restricts the maximum number of bytes we may copy. tc - a symbolic variable representing the number of times the instructions in I are executed
strlen	(I, tc, buf)	I - a set of instructions that together, perform the `strlen` operation. tc is a symbolic variable representing the number of times the instructions in I are executed. buf is the address of the frst byte of the buffer whose length we are calculating

Table 3: Summary of output of each semantic block analysis

should be re-executed, then the user can control the number of bytes copied into the destination buffer, up to the size of the source buffer. Alternatively, if we detect a block of code that would otherwise be Bounded Copy, if not for the *influence* that the user has over the bounds, then the block of code is also classified as Unbounded Copy. This corresponds to the example shown in Figure 6, which is really an unbounded copy, despite using `strncpy`, a bounded-copy function, since the user has control over the bounds.

Bounded copy Detecting a block of code that performs Bounded Copy is identical to detecting a block of code that performs Unbounded Copy, with the exception that the *exit conditions* of the loop includes an additional *exit condition* that controls the number of iterations that the loop makes, and which the user does not have control over. Let us make this notion more precise. Recall loop-extended symbolic execution (§ 5.2) extends symbolic execution to account for loops by introducing a symbolic *trip-count* (tc), which represents the number of times the loop is executed. Then the additional exit condition for a bounded copy is the predicate $tc >= c$ for some constant c, for which the user has no *influence* over, i.e., if c is from byte i, then $f_i(x) = \perp$. Otherwise, if $f_i(x) \neq \perp$, the user has *influence* over the byte c then this block of code is classified as an Unbounded Copy.

`strlen` We define the template $T_{\texttt{strlen}} = (\langle A := mem[src], src++\rangle, \{A, src, dest\}, \emptyset)$. If the block of code matching this template lies in a loop, and the *exit condition* of the loop matches the *exit condition* of an Unbounded Copy, then this block of code is semantically equivalent to `strlen`. Then, we add the constraint $tc = length(i)$, where i is the *buffer id* of the source buffer into the global Path Constraints. $length(i)$ corresponds to the *auxiliary attributes* introduced by Saxena et al [17].

5.4 Buffer inference

Buffers are data structures commonly used in programs to store strings of user input. Common operations associated with buffers are unbounded/bounded copying of buffers from place to place and `strlen`, which calculate length of buffers. Buffer overflow occurs when the copying of buffers causes sensitive memory regions to be overwritten with user data. If the destination and source buffers are of fixed size, then static

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  char buf[255];
  int len;

  if (argc < 3) {
    return 0;
  }

  len = atoi(argv[1]);
  strncpy(buf, argv[2], len);
  return 0;
}
```

Figure 6: Even though `strncpy` is a bounded-copy function, if the user controls the bounds, then this piece of code is semantically equivalent to an Unbounded Copy

analysis will quickly reveal the bug. Hence, in AHEG, we turn our attention to variable-sized buffers. Many buffer overflow occurs because the source buffers are variable-sized and the destination buffers, fixed-size, and so it is possible for the source buffer to be greater in size than the destination buffer. The key insight in detecting variable-sized buffers is to observe that by nature of the buffer being variable in size, the program does not know the size of the buffer beforehand, and hence operations on the buffer must involve a loop. In AHEG, we use this insight to identify 2 possible ways buffers propagate through the program: either via Bounded Copy or Unbounded Copy. Another operation performed on variable-sized buffers involving a loop is `strlen`.

A *buffer* object is a data structure with 5 fields, namely *buffer_id*, *actualLength*, emphpotentialLength, *srcBuffer*, *srcBufferOffset*. The *buffer_id* is also the address of the first byte of the buffer. *actualLength* refers to the number of concrete bytes in this buffer. *potentialLength* refers to the number of bytes that could be in this buffer, i.e. there exists a path P through this program such that this buffer will contain *potenialLength* many bytes. Since buffers propagate via copying, every buffer must come from some where; the first buffer in the program must come from either `SymFile` or `SymArgv`. Hence, *srcBuffer* refers to the *buffer_id* of the source buffer. It is not necessary that we start copying from the first byte of the source buffer to the destination buffer. If we had started from the n^{th} byte, then *srcBufferOffset* will be set to n.

Buffer introduction The only way for user input to enter the system is via the user input instructions, namely `ReadFile` and `ReadArgv`. When either instruction is encountered, we create a new *buffer* object. Recall the arguments to `ReadFile` and `ReadArgv` are *buf*, *NumBytesToRead* and *&NumBytesRead*. We set *actualLength* to the value contained in the address of *&NumBytesRead* and *potential length* to *NumBytesToRead*, if the user does not have any *influence* over it, and min(*NumBytesToRead*, range($f_i(x)$)), where i is the address of *NumBytesToRead*, otherwise. Notice that in this case, since *potentialLength* depends on f_i, *potentialLength* becomes symbolic. The *buffer_id* of this *buffer object* is the address at which this buffer is residing, namely *buf*. *srcBuffer* is set to either `SymFile` or `SymArgv` and *srcBufferOffset* is set to 0.

Buffer propagation Buffers propagate via Unbounded Copy or Bounded Copy. When we detect an Unbounded Copy, we clone the source *buffer* object and update the *buffer_id* of the clone with the destination address of the Unbounded Copy. When we detect a Bounded Copy, we create a new *buffer* object, set its *buffer id* to the destination address of the Bounded Copy, set the *actualLength* to the length of bytes actually copied into the buffer, and set its *potentialLength* to be the minimum of the bounds in the Bounded Copy and the *potential length* of the source buffer.

5.5 Example

A small example now might help us to see how all the algorithms presented in this section integrate together. Consider the line `strlen(buf)`, which semantically, computes the length of `buf`. We shall show how `AHEG` detects that `strlen(buf)` actually computes the length of `buf`.

Figure 7 and 8 shows 2 common techniques of calculating the length of a string. Figure 9 shows a snippet of the translation of Figure 7 into Simple ASM. Our loop detection analysis gives us the set $I = \{2,3,4,5,6,7,8,9,10,11\}$ of instructions in the loop. From instruction 3, since `edx` is dereferenced, we guess `edx` might be a pointer. The address that `edx` is pointing to matches the *buffer id* of a *buffer* object, so we tag the byte at `(ebp) - $8` with `PTR(buffer_id)`. Let's call this buffer *buf*, so `eax` is set to `buf[0]`. Instruction 4 check that `buf[0]` is not NULL. Otherwise, it jumps to line 12, which isn't in I. Hence, `buf[0] = NULL` is a exit condition for this loop. Also, $\forall i <$ length(buf), user has *influence* over `buf[i]`. Hence, this satisfy the *exit condition* requirement for an Unbounded Copy. It isn't classified as an Unbounded Copy, because the block of code doesn't match the template for Unbounded Copy. In particular, the code does not match $mem[dest] := A; dest++$. Nonetheless, we impose an additional constraint into *PC* that $tc =$length(buffer_id). Finally, the value of `[(ebp) - $4]` is 0, before the start of the loop, and since `ADD` is the only non-idempotent instruction, at the end of the loop, the byte at `(ebp) - $4` will be tagged with `REP(0, ADD $1, tc)`. Hence, the variable i, which resides in memory address `(ebp) - $4` is encoded as having the value of $0 + (tc \times 1)$, where $tc =$length(buffer_id), and we see immediately that length of strings are really just a special case of symbolic trip counts.

The case where `strlen` is computed with the second technique (Figure 8) is very similar to the case above. Instead of i, we encode `ptr2` with `REP(argv[1], ADD $1, tc)`, where argv[1] is the value of `ptr2` before the start of the while loop. Hence, `len` will be encoded as having the value `REP(argv[1], ADD $1, tc) - argv[1]`. Experiments show that `strlen` as implemented in the Microsoft C Runtime Library uses a variant of the second technique.

6 Heuristics

In this section, we propose 2 heuristics to search paths in the programs that are more likely to be exploitable.

6.1 Maximum buffer heuristic

We assume that the program adopts the standard C calling convention, where in the function prolog, we subtract a certain offset from the stack pointer to allocate memory on the stack. We keep track of all instructions `SUB ESP, $c` and `ADD ESP, $-c`, where c is a constant and find the maximum of all such c. We output this c.

6.2 Potential Exploit heuristic

Whenever we encounter a jump that dereferences memory directly, i.e. `UncondJump [mem]` or `CondJump cond, [mem]`, we look for the buffer closest to mem. Note that if mem contains user input, then this is a considered a bug. We then check if the distance between that buffer and mem is most its *potential length*. If it is, then we have found a potential buffer overflow. The heuristic will then impose an additional con-

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  char *buf;
  int i = 0;

  if (argc < 2) {
    return 0;
  }

  buf = argv[1];
  while ((*buf) != '\0') {
    buf++;
    i++;
  }

  printf("Length of string is %d\n", i);
  return 0;
}
```

Figure 7: We calculate strlen of a string, by introducing a new variable *i*, and increment the variable in a loop that steps through a string until we encounter a NULL byte. *i* contains the length of the string

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  char *ptr1;
  char *ptr2;
  int len;

  if (argc < 2) {
    return 0;
  }

  ptr1 = ptr2 = argv[1];

  while ((*ptr2) != '\0') {
    ptr2++;
  }

  len = (int)(ptr2 - ptr1);

  printf("Length of string is %d\n", len);

  return 0;
}
```

Figure 8: We initialize 2 pointers, `ptr1` and `ptr2` to point to the start of a string. We then increment `ptr2` until we encounter a NULL byte. We then subtract `ptr2` from `ptr1` to get the length of the string

```
1. MOV [(ebp) - $4], $0;
2. MOV edx, [(ebp) - $8];
3. MOV eax, [edx];
4. CondJump EAX, $0, EQ, 12;
5. MOV ecx, [(ebp) - $8];
6. ADD ecx, $1;
7. MOV [(ebp) - $8], ecx;
8. MOV edx, [(ebp) - $4];
9. ADD edx, $1;
10. MOV [(ebp) - $4], edx;
11. UncondJump 2;
```

Figure 9: Snippet of the Simple ASM translation of the code in Figure 7. *i* is stored in [(ebp) - $4] and *buf* is stored in [(ebp) - $8].

straint that the length of this buffer is at least the distance to mem, call this constraint c, to get PC', i.e. $PC' := PC \wedge c$, and output PC'. This heuristic was first proposed by Xu et al [21] who argued that we can detect buffer overflows by treating the length of buffers as symbolic and storing only *prefixes* of buffers, as opposed to *full buffers*. The *prefix* corresponds to the concrete data we have in the buffer and the length of this *prefix* corresponds to the field *actual length* in our *buffer* object. The length of the *full buffer* corresponds to *potential length*. However, unlike Xu et al's technique, our heuristic does not require annotations from the programmer.

7 Exploit Generation

Brumley et al [2] showed that exploits are a subset of the input accepted by a program and that we can automatically generate such exploits by finding the right constraints and solving for them. Avgerinos et al [1] explicitly constructed such constraints for stack-based buffer overflow exploits and format-string exploits. With Memory Tagging and SimpleASM, we will now generalize Avgerinos et al approach to explicitly construct constraints that describe heap exploits. Note that AEG actually generated shellcode, so that the output of AEG was a string that if run against the program will produce a shell. In AHEG, we do not produce shellcode, since we stop as soon as we can set EIP to arbitrary values. Thus the output of `AHEG` are really *crashing input*. We leave the generation and placement of shellcode to future work.

7.1 1-step exploits

Suppose that during the *Bug-find* phase, we found a path P such that EIP contain user input, say x. Via memory tagging, we get that $x = f(y)$, where $y \in A$, is an input that the program accepts. Recall that A is the input space of the program. The goal now would be to set x to arbitrary values, say 0xdeadbeef, and solve for $y' \in A$ such that $f(y') = 0xdeadbeef$ and running the program with y' will go down the same path P. To do that, we assert that $x = 0xdeadbeef$ with the rest of the path constraints of path P. This is similar to the approach taken by Avgerinos et al.

7.2 2-step exploits

The utility of the machinery we have developed becomes more apparent when we consider 2-step exploits. Consider the program shown in Figure 10. This program was given as an example on by-passing stack canaries. By overflowing `buf`, we can overwrite `ptr` to point to arbitrary memory, so that in the next `strcpy`, we achieve arbitrary write to arbitrary memory. `AHEG` provides the language express such exploits.

For every instruction that writes to memory, i.e. the destination operand of a data flow instruction uses the macro $[y]$ for some address y, we check for user's *influence* over y. If the user has *influence* over y and the source operand, say x, then that's almost an exploit. The next step would be to set y to an address that would be dereferenced by a control flow instruction. So, upon discovery that user has *influence* over y, we set a flag, and for all subsequent control flow instruction that dereferences memory at address z, we check if the formula $PC \wedge z = y \wedge x = 0xdeadbeef$ is satisfiable, where PC is the current path constraints. If it is, we output $PC \wedge z = y \wedge x = 0xdeadbeef$, and check that the input satisfying that constraint actually does write $0xdeadbeef$ to the address z, and hence cause EIP to be overwritten with $0xdeadbeef$.

To see how this relates to the example, suppose we have found a path P, such that we overflow `buf`. Then in the second `strcpy`, we are copying the buffer at `inp2`, which we control, to the buffer that `ptr` is pointing to. At a high level, `strcpy` has to dereference `ptr` in order to perform the copy. When that happens, we realise user has *influence* over `ptr`, because of the overflow. Later, when the function returns, it pops the return address off the stack and jumps to that address. `AHEG` detects that and immediately tries the constraint $PC \wedge$ `(ptr = return address of foo)` $\wedge$ `(inp2[0] = 0xef)` $\wedge$ `(inp2[1] = 0xbe)` $\wedge$ `(inp2[2] = 0xad)` $\wedge$ `(inp2[3] = 0xde)`, which is satisfiable and generates the input that sets EIP to

Program	Maximum Buffer Heuristic	Potential Exploit Heuristic	EIP Overwritten
Toy 1	1308	20	Yes
Toy 2	1308	93	Yes

Table 4: List of toy programs and the length of input to crash the program, as generated by the MaximumBuffer heuristic and Potential Exploit Heuristic. Note that the length generated by the Potential Exploit Heuristic is minimum; the program cannot be exploited with fewer bytes

$0xdeadbeef$.

```
void foo(char *inp, char *inp2) {
  char *ptr;
  char buf[16];
  strcpy(buf, inp);
  strcpy(ptr, inp2);
}
```

Figure 10: Write-through-pointer exploit. Code taken from http://www.ece.cmu.edu/ dbrumley/courses/18732-f11/slides/0919_c-memory-safety.pdf slide 12

8 Implementation

We implemented the ideas presented in this paper in an end-to-end system, AHEG, which when given the compiled binary of a buggy program running in Windows XP SP3, will generate a crashing input against the program. AHEG consists of 2 components, the main instrumentation component, and the solver, both written in C. We used Pintools [13] in the main instrumentation component to instrument the x86 instructions in order to dynamically translate x86 machine code into SimpleASM. We used Z3 Theorem Prover [9] for constraint solving.

9 Evaluation

9.1 Testing the main instrumentation tool

Appendix A shows a list of programs to test different aspects of the instrumentation tool. Figure 11 shows a number guessing game, and AHEG successfully generates 3 different inputs to explore all 3 states of the program. Figure 12 depicts a program which requires AHEG to reason about the length of user input in addition to the content of user input. AHEG generated an input of length 1, and an input of length 28. Figure 13 and 14 shows 2 programs with buffer overflow vulnerabilities. Both Maximum Buffer Heuristic and Potential Exploit heuristic successfully produced crashing input against both samples. Finally, the program in Figure 15 uses bounded copying to avoid the buffer overflow vulnerability. AHEG recognized that the copying was bounded and hence does not generate a false positive.

9.2 Comparing heuristics

We compared *Maximum Buffer Heuristic* with *Potential Exploit Heuristic* to determine the effectiveness of each heuristic. In all the experiments below, except RMMp3, the seed input given was ”hello”. For RMMp3, the seed input given was ”http://hello”.

Program	Size of program (in KB)	Length of crashing input (in bytes)	Time taken	EIP Overwritten
FatPlayer	800	4124	46.3 s	Yes
Mp3ToWave	1092	5420	176.7 s	No
CCMPlayer	714	4092	64.2 s	No
RMMp3	896	65860	58.3 s	No

Table 5: List of programs tested on `AHEG` and their size, the length of input to crash the program, as generated by the MaximumBuffer heuristic, and the time taken to generate the crashing input

Table 4 shows how the 2 heuristic compares on 2 toy programs - toy 1 (source code shown in Figure 13) and toy 2 (Figure 14). Both heuristics generated a string that caused EIP to be overwritten with user input. However, notice that *Potential Exploit Heuristic* generated a string that was much shorter than the string generated by *Maximum Buffer Heuristic*. In fact, *Potential Exploit Heuristic* generated a string that was minimum; inspection of the source code shows that we cannot get full control of EIP with fewer bytes.

Unfortunately, as *Potential Exploit Heuristic* assumes accurate *Buffer Inference*, which in turn assumes accurate *loop detection* and detection of *exit conditions*, *Potential Exploit Heuristic* failed to find an exploit in the real world programs, because 1) our definition of loop was not general enough and missed certain loops 2) we failed to detect several exit conditions which together caused `AHEG` to miss blocks of code which were performing Unbounded Copy. As a result, we were only able to use the *Maximum Buffer Heuristic* on real world programs.

Table 5 shows a list of programs known to have buffer overflow vulnerabilities. We tested these programs on `AHEG`, for which we successfully generated the crashing input. The length of the crashing input, as generated by the Maximum Buffer heuristic, is shown next to each program in the table. We generated these strings in less than 5 minutes, despite the large size of the programs, and the length of the crashing input.

10 Discussion

10.1 Undecidability

Claim: Solving AEG in general is undecidable

Proof: Suppose we had an oracle O that given any program, answers yes if the program is exploitable and no otherwise. Consider the program $P =$ if ($O(P)$ is yes) then loop forever else (buffer overflow vulnerability here). A program that does not depend on user input is defined to be not exploitable because an exploit is defined to be a subset of user input. In this example, P is self-referential in the sense that it was able to consult the oracle if it was exploitable. This is implementable, as evidenced by existence of quines.

10.2 Circumventing undecidability

Programs in finite state machines are decidable. We take advantage of the fact that processors are inherently finite state machines, since we have finite memory, albeit a very large one. This is the motivation for memory tagging. We use a finite state machine with more states to analyze programs written for finite state machines with smaller states, since we tag each byte with our own data. For example, we can imagine `AHEG` analyzing 32-bit programs on a 64-bit machine, and for each 64-bit word in memory, 32-bits belong to the program, and 32-bits are tag information.

11 Conclusion

In this paper, we developed Simple ASM, a simpler subset of x86 ASM, on which we develop algorithms such as *Buffer Inference*, *semantic analysis*, etc, which form the basis for *Memory Tagging*. We then showed how *Memory Tagging* can be used together with *Potential Exploit Heuristic* to generate minimum-length crashing input. We also showed how the tags in *Memory Tagging* can be used to construct the constraints describing a "2-step exploit", a common exploit technique against doubly-linked link list and often used in heap exploits. Finally, these ideas were implemented in an end-to-end system, `AHEG`, which analyzed 4 different programs compiled for Windows XP SP3 and generated crashing input against them in under 5 minutes.

References

[1] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. Aeg: Automatic exploit generation. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2011.

[2] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2008.

[3] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Symposium on Operating System Design and Implementation*, 2008.

[4] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the International SPIN Workshop on Model Checking of Software*, 2005.

[5] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the ACM Conference on Computer and Communications Security*, Oct. 2006.

[6] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-aware malware detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2005.

[7] L. Clarke and D. Richardson. Symbolic evaluation methods for program analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981.

[8] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis*, pages 196–206, New York, NY, USA, 2007. ACM.

[9] L. de Moura and N. Bjrner. Z3: An efficient smt solver. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2008.

[10] huku. Yet another free() exploitation technique. *Phrack*, 13(66), 2009.

[11] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the Third International Workshop on Automated Debugging*, 1995.

[12] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19:386–394, 1976.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2005.

[14] T. Moseley, D. Grunwald, D. A. Connors, R. Ramanujam, V. Tovinkere, and R. Peri. Loopprof: Dynamic techniques for loop detection and profiling. In *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA) held in conjunction with ASPLOS-12*, October 2006.

[15] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2005.

[16] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2004.

[17] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution of binary programs. In *International Symposium on Software Testing and Analysis*, 2009.

[18] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, Feb. 2000.

[19] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[20] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering*, 2005.

[21] R.-G. Xu, P. Godefroid, and R. Majumdar. Testing for buffer overflows with length abstraction. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 27–38, New York, NY, USA, 2008. ACM.

```c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  int x;
  int magic_number = 6;

  if (argc < 2) {
    printf("Usage: %s number\n", argv[0]);
    return 1;
  }

  x = atoi(argv[1]);

  if (x > magic_number) {
    printf("Too big\n");
  }
  else if (x < magic_number) {
    printf("Too small\n");
  }
  else {
    printf("You got it\n");
  }

  return 0;
}
```

Figure 11: A number guessing game with 3 possible states - too high, too low, and correct

A Appendix

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
  if (argc < 2) {
    printf("%s string", argv[0]);
    return 0;
  }

  if (strlen(argv[1]) == 28) {
    printf("Very good\n");
  }
  else {
    printf("Not very good\n");
  }

  return 0;
}
```

Figure 12: 2 possible states - one where length of user input is 28 bytes and one where it isn't

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void toad() {
  printf("You got into toad!\n");
}

int main(int argc, char *argv[]) {
  void (*funPtr)() = toad;

  char smallbuf[20];
  if (argc < 2) {
    printf("%s string", argv[0]);
    return 0;
  }

  //strcpy(smallbuf, argv[1]);
  strcpy(smallbuf, "blahblah");
  strcat(smallbuf, argv[1]);

  funPtr();

  exit(1);

  printf("dead code\n");

  return 0;
}
```

Figure 13: A program with a buffer overflow vulnerability. Program gets input from command line.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void toad() {
  printf("You got into toad!\n");
}

int main(int argc, char *argv[]) {
  FILE *fp;
  fp = fopen("input", "r");

  void (*funPtr)() = toad;

  char smallbuf[100];
  char bigbuf[255] = {0};

  strcpy(smallbuf, "Blah Blah Blah blah");


  fgets(bigbuf, 125, fp);
  strcat(smallbuf, bigbuf);
//  strcpy(smallbuf, bigbuf);

  printf("User input: %s\n", smallbuf);
  funPtr();

  exit(1);

  printf("dead code\n");

  return 0;
}
```

Figure 14: A program with a buffer overflow vulnerability. Program gets input from file.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
  char buf[255];

  if (argc < 2) {
    printf("%s string", argv[0]);
    return 0;
  }

  strncpy(buf, argv[1], 255);

  printf("User input: %s\n", buf);

  return 1;
}
```

Figure 15: Bounded copying is used to avoid buffer overflow vulnerability