Finding Buffer Overflow Inducing Loops in Binary Executables

Sanjay Rawat*
Verimag Lab
University of Grenoble, Grenoble, France
Email: Sanjay.Rawat@imag.fr

Laurent Mounier
Verimag Lab
University of Grenoble, Grenoble, France
Email: Laurent.Mounier@imag.fr

***Abstract*—Vulnerability analysis is one among the important components of overall software assurance practice. Buffer overflow (BoF) is one example of the such vulnerabilities and it is still the root cause of many effective attacks. A general practice to find BoF is to look for the presence of certain functions that manipulate string buffers, like the `strcpy` family. In these functions, data is moved from one buffer to another, within a loop, without considering destination buffer size. We argue that similar behaviour may also be present in many other functions that are coded separately, and therefore are equally vulnerable. In the present work, we investigate the detection of such functions by finding loops that exhibit similar behaviour. We call such loops Buffer Overflow Inducing Loops (BOIL). We implemented a lightweight static analysis to detect BOILs, and evaluated it on real-world x86 binary executables. The results obtained show that this (simple but yet efficient) vulnerability pattern happens to be very effective in practice to retrieve real vulnerabilities, providing a drastic reduction of the part of the code to be analysed.**

***Index Terms*—Buffer overflow, security vulnerability, dependency chain, loop detection, static analysis, binary code.**

I. Introduction

Vulnerability detection techniques, and more particularly buffer overflow detection techniques (which is still reported as 3rd most dangerous software error [1]), have been already widely addressed by the research community. Roughly speaking, the proposed techniques can be classified into two categories:

- The static approaches, consists in performing some code analysis (usually based on data-flow analysis or abstract interpretation), without executing the application. The objective here is to detect potential programming flaws, that are known to be vulnerability prone. It could range from taint-dependency analysis (to detect that a user input can be written into a buffer), to more sophisticated value-analysis (to detect that a buffer can be accessed out of its bound). The success of these approaches essentially depends on the trade-off they provide between the execution cost (i.e., how precise is the analysis) and the number of false positives they give. Typical existing tools are CodeSurfer[2] and Parfait[3].
- The dynamic approaches, consists in executing a (monitored) instance of the application, in order to "reveal" some unexpected behaviour. This category includes popular fuzzing techniques enhanced with dynamic symbolic executions, or genetic-based testing. Since only a part of the application can be exercised, a difficulty here is to choose the inputs that are more likely to trigger some vulnerability at execution time. Typical existing tools are BitBlaze[4], SAGE[5], Dytan[6], etc.

Work is funded by ITEA2 DIAMONDS project.

A natural trend is to try to combine these two categories in order to "guide" the dynamic analysis towards some potential flaws that have been detected statically. This can be done for instance by identifying some *vulnerability patterns*, corresponding to pieces of code containing potentially vulnerable instructions.

A. Vulnerability patterns

A simple solution consists in defining vulnerability patterns at the syntactic level. This is the case for instance when focusing the search on well-known vulnerability prone functions. A typical example is the `strcpy` family of C/C++ functions, that are notorious for buffer overflow vulnerabilities. As a result, many dynamic vulnerability analysis studies consider known library functions as target (or sink) for tainted data flow (e..g [7]). However, the main culprit for the introduction of the vulnerability it is not the function itself, but the way this function manipulate data.

For example, `strcpy` function copies the source string buffer to destination string buffer without considering the length of the destination buffer. This data movement is done within a loop based on the length of the source buffer. However, such a behaviour can occur in many other functions, not belonging to the `strcpy` family.

The objective of this work is to consider more of "semantic" definition of vulnerability patterns, based on some criteria that can be checked by means of lightweight static analysis, from binary code. Note that similar views are also presented in [8], but to the best of our knowledge, there is no formalism and implementation presented so far.

To make this position more precise, let us consider the function shown in listing 1:

In this function, `Src` and `Dest` are pointers to the source and destination buffer which are passed to the function as arguments. A local variable `*p` is defined, and its value is set as `Dest`. There is a *while* loop that iterates over `Src` and

```
char * bufCopy(char *Dest, char *Src)
  {
      char *p = Dest;
      while (*Src != '\0')
      {
          *p++ = *Src++;
      }
      *p = '\0';
      return Dest;
  }
```

Listing 1. Example of a function that is similar to strcpy

moves the corresponding characters to `p` (i.e., to the destination). Functionality wise its behaviour is similar to `strcpy()` function. Therefore, while performing a vulnerability analysis, if we look only for calls to `strcpy` family, we may be missing several other vulnerable functions.

Several features in this code are likely to make this function vulnerable (aka *bad code smell*):

- it contains a loop;
- the loop involves a buffer traversal, which means that the memory location which is being read/written *is changing at each loop iteration*;
- the destination buffer written in the loop has been passed as a function argument, which makes its size unknown inside this function;
- the source buffer is also passed as a function argument, which makes this content/length non controllable inside the function (it may be derived from user inputs without any sanity checks);
- the number of loop iteration is neither fixed, nor does it depend on the destination buffer.

In the next section we define more precisely which of these features we could consider in our context to classify this function as vulnerable.

This observation is in conformance with Microsoft C/C++ compiler's definition of vulnerable function (*The definition of a vulnerable function is one that allocates a type of string buffer on the stack.*) when inserting `/GS` flag[1]. However, we feel that the above definition is too conservative in the sense that mere the presence of buffer does not make it vulnerable, but there should be some *interesting* operation on this.

B. *Defining* BOP*s and* BOIL*s*

The features that make a strcpy-like function potentially vulnerable cannot be addressed by lightweight static analysis. Indeed, most of them are even undecidable in general. As a result, we have to restrict ourselves to some simpler criteria, while still able to reflect potential vulnerabilities.

From this point of view, we propose that a function is worth analysing rigorously if it involves **B**uffer **O**verflow **I**nducing **L**oops (BOIL). We call such a function **B**uffer **O**verflow **P**rone (BOP) function. A loop is a BOIL if there is memory write within the loop **and** that memory is changing within the loop.

[1]http://msdn.microsoft.com/en-us/library/8dbf701c(v=vs.80).aspx

This is a very loose definition and we may include other constraints like the number of iterations should not be fixed or should not be based on destination buffer, etc.

In this work, we analyse the binary executable of the application to find BOP functions (i.e. functions with BOILs). The main motivation to analyse binary is manifold:

1) It is a general practice to use third-party's libraries those source code is not available.
2) There is significant difference between the source code of the application and the machine code ultimately run [9].
3) Multi-language applications that are complied to one common byte-code (e.g. .NET*ish* languages).
4) Analysing the executable file increase the scope of such analysis specially in the case of COTS softwares.

C. *Contribution and structure of the paper*

While the above definition of BOIL is simple and short, detecting such loops in binary executables is challenging. Apart from discovering the control structures in the executable, tracing the memory access imposes biggest hurdle. On the top of that, we observe that memory access and loop body structure are compiler dependent and compilers may not follow the same rules always. In the following sections, we elaborate such points and propose a method to detect such loops.

The rest of the paper is organised as follows: section II covers the basic problem and the way we formalise the solution. Implementation details are described in section III, which is followed by the section IV to add a small improvement to the main proposal. The section V provides details on experimental results. As a reference, some of the related work is provided in section VI. We conclude the article in section VII by summarising the current work and some future directions.

II. BOIL DETECTION

First, we give some intuition on how buffer-accessing loops are translated by the compilers. This will give us some (simple) criteria to detect such patterns in executable code. Then we formalise these criteria in terms of specific dependency graphs that can be extracted from loop bodies. These results will be used to develop the BOIL detection algorithm presented in section III.

A. *Understanding the Difficulty*

Let us consider an array `X` accessed in a loop body L, and assume that `c` is invariant inside L (its value is constant), `i` is the loop induction variable, `f` is a (side-effect free) function, and that `y` is an arbitrary value (that may change or not at each iteration). According to [10], write access patterns to `X` can be classified into three main categories:

- *Constant access*: the *same* memory location is accessed at each iteration, as in `X[c] = ....`
- *Stride access*: memory locations are accessed with a *fixed stride*, representing the distance between two successive write addresses, as in `X[f(i)] = ...`

```
1 004075F0   _strcpy
2 004075F0   push      edi
3 004075F1   mov       edi, ss:[ebp+Dest]
4 004075F5   jmp       loc_407661
5
6 00407661   mov       ecx, ss:[ebp+Src]
7 00407665   test      ecx, 3   —> exit the loop
8 0040766B   jz        loc_407686
9
10 0040766D  mov       byte dl, byte ds:[ecx]      <——
11 0040766F  inc       ecx                            |
12 00407670  test      byte dl, byte dl               |
13 00407672  jz        loc_4076D8  —> exit the loop   |
14                                                    |
15 00407674  mov       byte ds:[edi], byte dl         |
16 00407676  inc       edi                            |
17 00407677  test      ecx, 3                         |
18 0040767D  jnz       loc_40766D -> loop back to  ——
```

Listing 2. Assembly code of the strcpy function

```
1 00401000   bufCopy
2
3 00401010   mov       eax, ss:[ebp+Src]             <——
4 00401013   movsx     ecx, byte ds:[eax]              |
5 00401016   cmp       ecx, 0x0                        |
6 00401019   jz        loc_40103C  —> exit the loop    |
7                                                      |
8 0040101F   mov       eax, ss:[ebp+p]                 |
9 00401022   mov       ecx, eax                        |
10 00401024  add       eax, 0x1                        |
11 00401027  mov       ss:[ebp+p], eax                 |
12 0040102A  mov       eax, ss:[ebp+Src]               |
13 0040102D  mov       edx, eax                        |
14 0040102F  add       eax, 0x1                        |
15 00401032  mov       ss:[ebp+Src], eax               |
16 00401035  movsx     eax, byte ds:[edx]              |
17 00401038  mov       byte ds:[ecx], byte al          |
18 0040103A  jmp       loc_401010 —> loop back to  ——
```

Listing 3. Assembly code of the loop body of function `bufCopy`

- *Non-monotonic*: memory locations are accessed in a random way, or at least the access pattern is based on some complex expression, that cannot be easily identified from a static point of view, as in `X[f(y)] = ...`

Clearly, item 1 cannot lead to a buffer overflow (with respect to buffer `X`), and item 3 is not regular enough to be analysed in an automated way. Moreover, such memory access is less likely to be interesting for buffer overflow based exploits. On the other hand, item 2 is commonly encountered and it corresponds to a typical array traversal situation. In the following we will consider only this later case and, therefore, focus on *stride memory accesses*.

At the source level, stride memory accesses corresponding to array traversals are most of the time encoded using address expressions relative to the *previously visited element*, as in listing 1 (`*p++ = ...`). To better understand how such high-level access patterns are translated into binary code, let us consider two listing of assembly code: an excerpt of the code of `strcpy` (listing 2), and an excerpt of the code produced from function `bufCopy` (listing 3). These assembly codes have been slightly modified to make them more readable (see below).

Let us first consider the listing 2. Function `strcpy()` takes two arguments, `Dest` and `Src`, indicating the *addresses* of the destination and source string buffers respectively. Arguments and local variables are stored on the execution stack. Their addresses on the stack are expressed as offsets with respect to the base pointer, noted `ebp` in x86 architectures. In this listing, `ebp+Dest` (resp. `ebp+Src`) denotes the address of *Dest* argument (resp. of *Src* argument), whereas `[ebp+Dest]` (resp. `[ebp+Src]`) denotes the memory location pointed to by `ebp+Dest` (resp. `ebp+Src`), namely the addresses of the first character of the destination buffer (resp. of the source buffer). The loop body (copying each source character into the destination) lies between lines 10 and 18 (the arrow from line 18 to 10 indicates the loop back-edge). In terms of memory accesses, the code detail can be sketched as follows:

- the addresses of the first characters of `Dest` and `Src` buffers are copied into registers `edi` and `ecx` (lines 3 and 6), outside the loop body;
- the character pointed to by `ecx` (namely the current `Src` element) is copied into register `dl`, then `ecx` is incremented in order to point to the next source character (lines 10-11);
- the content of `dl` is then copied into the memory location pointed to by `edi` (namely the current `Dest` location), then `edi` is incremented in order to point to the next destination location (lines 15-16).

Let us now examine listing 3, corresponding to the loop body of function `bufCopy` (described in listing 1). This function takes two arguments `Dest` and `Src`, and uses one local variable (pointer `p`), whose addresses in the execution stack are `ebp+Dest`, `ebp+Src` and `ebp+p` respectively. Initially `p` is assigned with `Dest`, this is not shown in listing 3. The copy operation between `Src` and `Dest` is more complex than in the previous example. It can be summarised as follows:

- the address of the current location pointed to by `p` is copied into `eax` (line 8), then moved to `ecx` (line 9), `eax` is incremented and written back to `p` which now stores the address of the next destination location (lines 10-11);
- the same scenario occurs for buffer `Src`: the address of the current source character is copied into `edx` (lines 12-13), and the address the next source character is assigned to `Src` (lines 14-15);
- finally, the current source character (stored in `edx`) is written to the current destination location (lines 16-17);
- lines 3-6 evaluates the loop condition, to exit if the current source character is equal to 0.

Recall that BOIL detection means identifying loops containing at least *memory writes*, at *changing addresses*, and *depending on local variables and arguments*, the comparison between these two codes raises several remarks:

- In listing 3, the dependencies among memory location read or write inside the loop and the local variables and arguments are directly visible inside the loop body. It is not the case for listing 2.
- In listing 2, within the loop, the memory is written only once (at line 15). In listing 3, within the loop, memory

is written 3 times - two times for changing the stored address of the next character (at lines 11 and 15) and one time for storing the character itself (at line 17).

- As a consequence, in listing 3, the memory address change is also done via a memory write. In other words, *memory write operation is involved in the process of changing the address of the memory that is being written within the loop*. Whereas, in the case of listing 2, memory address change is a matter of incrementing the register that contains the address of the memory that is being written.

We find these two compilation patterns stable across few tested compilers (MS C/C++, GCC, etc) and in the following we will call them **pattern A** (listing 2) and **pattern B** (listing 3).

B. Formalisation of the Solution

We now try to formalise more the common features of the code patterns discussed in the previous section. The main objective here is to provide some simple (but general) criteria to be used for BOIL detection.

First, the so-called stride memory access mode gives rise to a general pattern inside the loop body of the following style:

$$MEM_{cur} = MEM_{pre} + stride \qquad (1)$$

where, MEM_{cur} is the current memory location and MEM_{pre} is the previous memory location, and *stride* is a constant which may depend on the type of the buffer elements.

To interpret equation 1 at the assembly level, the following facts have to be taken into consideration:

- Memory write operations and address computations are distinct instructions, i.e., equation 1 is written as
```
adr = adr + stride
MEM[adr] = ...
```
- Since we focus on *stack-based* buffer overflow detection we assume that address `adr` is either contained in a register, or it results from some combination of the base register `ebp` and some offset.

As already mentioned, a loop will be classified as BOIL if its body contains a memory write to a changing address `adr`. Thus, a necessary condition is that the value written into `adr` within the loop body depends on `adr` itself (like in `adr = adr + stride`), or, at least, on another changing value inside the loop body. This can be expressed in terms of (self)-dependency chains computed from `adr`, as explained below.

A *use-definition chain* [11] (or *UD chain*) of a given instruction s is a set of instructions *defining* a variable *used* in s. We slightly generalise this definition to the notion of *backward dependency chain*: a backward dependency chain for a variable v used in an instruction s is a sequence of variables *that have been successively defined* (on a given execution) to define v. To formalise this notion, we introduce some definitions based on the (classical) notion of *dependency chain*. We will adopt the following notations:

- $\mathcal{I} = [s_1, s_2, \dots s_n]$ is an execution sequence (where each s_p is an instruction);
- $\mathcal{K} = [s_{k_1}, s_{k_2}, \dots s_{k_m}]$ is called *a sub-sequence* of $\mathcal{I}$ if it contains some of the instructions of $\mathcal{I}$ taken in this order, namely (i) $\forall k \in [1, m].\, s_{k_i} \in \mathcal{I}$ and (ii) $k_i < k_{i+1}$;
- v denotes an arbitrary *memory location* (that could be either a register `r`, or a memory address of the form `[r]` or `[r+offset]`).
- For a given instruction s, Def(s) (resp. Use(s)) designates the set of variables defined (resp. used) by s (e.g., the *lhs* and *rhs* of an assignment).

a) Backward Dependency Chain: A backward dependency chain from v over $\mathcal{I}$ is a sequence of memory locations v_1, v_2, ..., $v_m = v$ such that $s_{k_1}, s_{k_2}, \dots, s_{k_m}$ is the *longest* sub-sequence of $\mathcal{I}$ verifying:

- $s_{k_m} = s_n$
- $v_i \in \text{Def}(s_{k_i})$
- $\text{Def}(s_{k_i}) \cap \text{Use}(s_{k_{i+1}}) \neq \emptyset$
- $\forall j \in]k_i, k_{i+1}].\, \text{Def}(s_j) \cap \text{Use}(s_{k_{i+1}}) = \emptyset$

We can now introduce the notion of *self-dependency chain* as a special backward dependency chain:

b) Self-Dependency Chain: A self-dependency chain over $\mathcal{I}$ is a backward dependency chain v_1, v_2, $\dots v_m$ s.t. $v_m = v_i$ for some $i \in (1, 2, \cdots, m-1)$.

In order to visualise the above definitions, we can distinguish two cases of self-dependency chain:

1) when $v_i = v_1$ (Fig. 1(i))
2) when $v_i = v_j, j \in (2, 3, \cdots, i-1)$ (Fig. 1(ii))

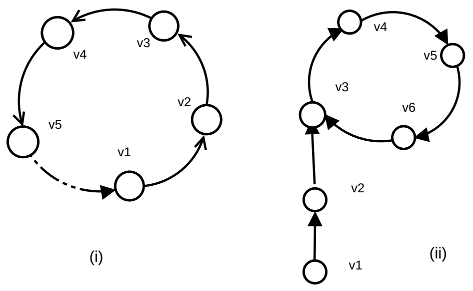

Fig. 1. Self-dependency chain graph shape.

Intuitively, the notion of self-dependency captures the fact that a variable v (or register/memory location) changes within the loop because its current value is obtained by something which is based on its previous value.

The shape of the dependency chain for memory location `edi` inside **pattern A** corresponds to Fig. 1(i). This is illustrated by the following excerpt of an execution sequence produced by listing 2 (where the loop is executed twice):

```
1: MEM[edi] ← SRC          2: edi ← edi+1
3: MEM[edi] ← SRC
```

However, for the case of **pattern B**, the change in the memory address written inside the loop does not appear as a (simple) self-dependency chain, as illustrated by the code fragment below (corresponding to an excerpt of listing 3):

```
1: reg ← MEM[base+offset]
```

```
2: reg1 ← reg
3: reg ← reg+stride
4: MEM[base+offset] ← reg
5: MEM[reg1] ← SRC
```

Indeed, in this example, `base+offset` does not change, but its content depends on `reg`, which changes inside the loop (line 3). So, here the "self-dependency" is not visible explicitly. In the figure 2, we can see that there is a dependency chain starting at `MEM[reg1]` and ending at `Mem[ebp+offset]`, and that there is a self-dependency chain starting at `MEM[ebp+offset]`. This situation can be captured by extending our previous notion of dependency chain as follows:

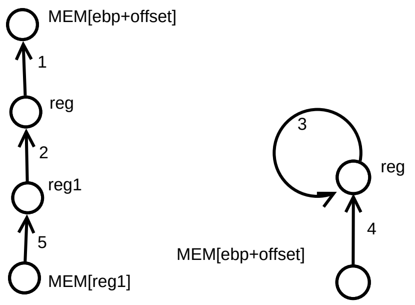

Fig. 2. Dependency chain for pattern B.

c) Extended Self-Dependency Chain: There exists an extended self-dependency chain from a memory location v over an execution sequence $\mathcal{I}$ if there exists a non-empty set of dependency chains $C_1 = [v^1_1, \ldots v^1_{n_1}]$, $C_2 = [v^2_1, \ldots v^2_{n_2}]$, $\ldots C_p = [v^p_1, \ldots v^p_{n_p}]$ such that:

- $\forall i \in [1, p-1].\ v^i_{n_i} = v^{i+1}_1$;
- $v^p_{n_p} = v$;
- C_1 is a self-dependency chain.

Figure 3 shows an example of extended self-dependency chain corresponding to **pattern B**.

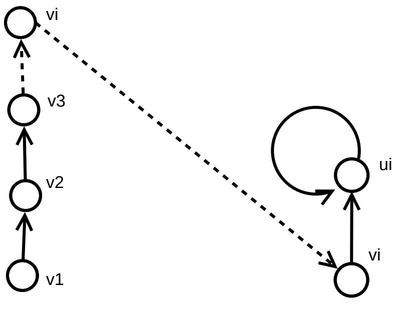

Fig. 3. Extended Self-Dependency chain.

Using the previous definitions, we can now state more formally the criterion we propose to use for BOIL detection.

d) Iteration dependent value: The value of a memory location v used in a loop depends on the number of iteration performed if there exists an execution sequence of the loop body $\mathcal{L}$ which contains an extended self-dependency chain starting from v.

The proof of this claim is obtained by contradiction: if there is no extended self-dependency chain starting from v, this means that, for all possible execution sequence of $\mathcal{L}^*$, v depends only on initial values defined outside the loop. As a consequence, the value of v does not depend on the number of iteration performed.

e) A BOIL detection criterion: A loop is classified as BOIL if it contains a write to an address which is iteration dependent (as defined below).

Of course, since this (static) criteria provides only a necessary condition we may get false positives, as illustrated in the following loop body:

1: adr = adr + 0 2: MEM[adr] = ...

Note that since we only deal with a special case of buffer overflows (corresponding to particular array accesses) we may get some false negatives as well. However, we will see in section V that this criterion happens to be accurate enough in many realistic examples.

III. IMPLEMENTATION DETAILS

In this section we discuss some implementation issues of BOIL detection based on the criterion stated above. First we present the framework we used to develop this solution, and how it impacts our work. Then we describe in more details the algorithmic part of our implementation.

A. The BinNavi framework

Our implementation is based on BinNavi [12], a reverse-engineering framework. BinNavi is based on the preliminary analysis done by the IDA Pro disassembler [13]. Starting from the assembly code produced by IDA Pro from binary executables, BinNavi provides several intermediate program representations based on an meta-assembly intermediate language called REIL. These intermediate representations (call graph, control-flow graph, etc.) can be accessed through Jython APIs to write static analysis modules.

From a prototyping point of view, one of the most interesting feature of BinNavi is the REIL intermediate language. This language is defined from 17 instructions only, with an uniform syntax, following a 3-address instruction format - `inst op1, op2, op3` - where `inst` is the opcode of the instruction, `op1` and `op2` are the operands, and `op3` is the resultant. Operands are either registers or immediate values, and the result is always a register.

The underlying CPU architecture is very simple as well (unlimited memory and unlimited number of registers). Of course, translation from the assembly source code is not one-to-one, but there is a correspondence between the offsets of native instructions and the offsets of REIL instructions. In addition, native register names (like `ebp`, `edi`, etc.) are preserved in the REIL code.

Translation to REIL is available from several codes (x86, ARM, PowerPC, and MIPS). Our experiments are based on x86 executables.

```
40101F00   add    0xFFFFFFFC, ebp, qword t0     // 0040101F
     mov eax, ss: [ebp + var_4]
40101F01   and    qword t0, 0xFFFFFFFF, t1
40101F02   ldm    t1, , t2
40101F03   str    t2, , eax
40102200   str    eax, , ecx                    // 00401022
     mov ecx, eax
40102400   and    0x1, 0x80000000, t0           // 00401024
     add eax, 1 ## *p++
40102401   and    eax, 0x80000000, t1
40102402   add    0x1, eax, qword t2
40102403   and    qword t2, qword 0x80000000, t3
40102404   bsh    t3, −31, byte SF
40102405   xor    t0, t1, t4
40102406   xor    t4, 0x80000000, t5
40102407   xor    t0, t3, t6
40102408   and    t5, t6, t7
40102409   bsh    t7, −31, OF
4010240A   and    qword t2, qword 0x0, qword t8
4010240B   bsh    qword t8, qword −32, byte CF
4010240C   and    qword t2, qword 0xFFFFFFFF, t9
4010240D   bisz   t9, , byte ZF
4010240E   str    t9, , eax
40102700   add    0xFFFFFFFC, ebp, qword t0 // 00401027 mov
       ss: [ebp + var_4], eax ## *p++
40102701   and    qword t0, 0xFFFFFFFF, t1
40102702   stm    eax, , t1
40102A00   add    0xC, ebp, qword t0           // 0040102A mov
     eax, ss: [ebp + arg_4]
40102A01   and    qword t0, 0xFFFFFFFF, t1
40102A02   ldm    t1, , t2
40102A03   str    t2, , eax
40102D00   str    eax, , edx                   // 0040102D mov
     edx, eax
40102F00   and    0x1, 0x80000000, t0          // 0040102F add
     eax, 1 ## *source++
40102F01   and    eax, 0x80000000, t1
40102F02   add    0x1, eax, qword t2
40102F03   and    qword t2, qword 0x80000000, t3
40102F04   bsh    t3, −31, byte SF
40102F05   xor    t0, t1, t4
40102F06   xor    t4, 0x80000000, t5
40102F07   xor    t0, t3, t6
40102F08   and    t5, t6, t7
40102F09   bsh    t7, −31, OF
40102F0A   and    qword t2, qword 0x0, qword t8
40102F0B   bsh    qword t8, qword −32, byte CF
40102F0C   and    qword t2, qword 0xFFFFFFFF, t9
40102F0D   bisz   t9, , byte ZF
40102F0E   str    t9, , eax
40103200   add    0xC, ebp, qword t0           // 00401032 mov
       ss: [ebp + arg_4], eax ## *source++
40103201   and    qword t0, 0xFFFFFFFF, t1
40103202   stm    eax, , t1
40103500   ldm    edx, , byte t0               // 00401035 movsx
       eax, byte ds: [edx]
40103501   xor    byte t0, byte 0x80, byte t1
40103502   sub    byte t1, byte 0x80, t2
40103503   and    t2, byte 0xFFFFFFFF, byte t3
40103504   str    t3, , eax
40103800   and    eax, byte 0xFF, byte t1 // 00401038 mov
     byte ds: [ecx], byte al
40103801   stm    byte t1, , ecx
40103A00   jcc    byte 0x1, , 0x401010  // 0040103A jmp
     loc_401010
```

Listing 4. Part of the REIL code obtained from listing 3

B. Developing the solution in REIL

When considering the REIL intermediate language, the BOIL detection technique we proposed in the previous section can be slightly refined. To illustrate this for **pattern B**, let us consider the REIL code, given in listing 4, which is produced from the x86 source of listing 1.

Once a loop has been detected in the code, the first step is to identify memory write instructions within its body. Although not so obvious at the x86 level (due to the syntax complexity), this task happens to be much simpler in REIL. Indeed, there are only two memory access instructions: `STM` (for "store memory") and `LDM` (for "load memory"). For our purpose, `STM` is the instruction that serves well. Its format is `stm src,,dest`, where `src` is a literal or register whose value is stored at the memory address contained in register `dest`. In listing 4, there are 3 `STM` instructions, at addresses 40102702, 40103202 and 40103801.

The second issue is now to check for the existence of extended dependency chains starting from the store addresses used by these STM statements. This address is contained in their third operand. For the instructions at 40102702 and 40103801, these `dest` registers (registers `t1` and `ecx` resp.) are both dependent on memory address `ebp + 0xFFFFFFFC`, at instructions 40102700 and 40101F0 resp. Further, if we trace back the `src` of the instruction at 40102702 (register `eax`), we find that it depends on itself, with the corresponding self-dependency chain being given below:

```
40102702   stm    eax, , t1
4010240E   str    t9, , eax
4010240C   and    qword t2, qword 0xFFFFFFFF, t9
40102402   add    0x1, eax, qword t2
```

Listing 5. Backward dependency chain of the `src` of a `STM` instruction

According to our criterion we can conclude that the content of the destination register of the STM instruction used at address 4010381 is changing at each iteration. Hence, this loop is classified as a BOIL. We detail below the algorithms we used to implement these steps.

C. Finding Loops

Loop detection is performed using the standard algorithm, given in [11]. This algorithm starts with calculating dominator tree of function's instruction graph and is based on the observation that any child of a node which also happens to be it's dominator, indicates the presence of a loop. BinNavi provides a direct API to calculate dominator tree. For each loop, we extract the control-flow graph of the loop body.

Thereafter, we analyse each loop separately, looking for STM statements. If the loop does not have any STM, it is clearly classified as "not interesting". Otherwise, we need to check more carefully if this loop is a BOIL.

D. *Identifying* BOIL*s*

The BOILs detection algorithm takes a loop as input and it returns true if and only if this loop is a BOIL.

As discussed in section II-B, this algorithm is based on backward dependency chain computations, distinguishing two different situations:

- self-dependency chains: an element of the chain depends on itself;
- extended self-dependency chains: the chain terminates either outside the loop body, or it reaches a variable starting a new (extended) self-dependency chain.

From the implementation point of view, this check is performed by starting a (backward) *def-use* chain computation from all instructions of the loop until one of the following conditions holds:

1) we reach an instruction involving an operand already present in the current chain ⇒ this chain is a self-dependency chain, this loop is a BOIL;
2) we reach an instruction involving `ebp` or `esp` ⇒ this chain may be an extended self-dependency chain, a new computation will start again from this operand;
3) we reach an instruction not included in the loop ⇒ this loop is not a BOIL.

This implementation uses the BinNavi API, allowing to access the control-flow graph of each function and to easily compute *def-use* chains.

IV. Extending Boil Definition

As mentioned earlier, our current definition of Boil does not consider all the required features to trigger a buffer overflow vulnerability. One of them is to ensure that *the loop iteration condition is dependent on source buffer*. In this section, we propose a simple, yet effective, mechanism to include this extra check. This makes the definition of Boils a bit stronger (i.e. less conservative).

Of course, we still want to limit ourselves to scalable static analysis techniques, and therefore the solution we propose here is not general. More precisely, we consider the case where the loop iteration is controlled by a specific variable, i.e., the induction variable is different than the variables related to source or destination buffers. This can be a local variable or one of the arguments passed to the function.

Thus, our underlying assumption is that the loop controlling variable is referred in the program as `ebp + offset`, where the offset *should* be different from that for source or destination buffer w.r.t. `ebp`. Following simple steps describe the extra check:

1) Find in the loop body a *conditional branch* instruction. In the case of REIL, the corresponding instruction is JCC, those syntax is `jcc reg1, ,reg2` where `reg1` is set according to a CMP instruction to decide if the jump is taken. Register `reg2` is the target address of the jump.
2) Check if `reg2` is an address outside of the loop. This is the condition to find loop controlling variable. We assume that the loop is a *standard* loop [14][2].
3) Perform a backward dependency chain on `reg1`, which may go beyond loop body instructions to reach an instruction which has `ebp` as operand.
4) For patterns A and B of main Boil detection algorithm, find the backward dependency chain for the source of corresponding `STM` instruction(which is already calculated for pattern B as described in section III-D). We calculate it for pattern A following the same approach.
5) This backward dependency chain stops at `ebp + offset` type instruction. If the `offset` is same for step 3 and step 4 dependency chain results, classify the loop as Boil.

[2]If there are multiple loop exits, we consider only the first one. In such case, we may produce a false positive as the corresponding `cmp/jcc` may not be loop terminating factor.

As can be seen from the above formulation, this check makes the Boils more specific. It should also be clear from this formulation that if a Boil involves a vulnerability which depends on the destination buffer, such a negative array index access in destination buffer (freeType vulnerability, explained in section V-A) , it will be missed (meaning that there are still false negatives). We will see in section V how this extension impacts our experimental results.

V. Experimental Results

Our work objective is to find so-called BOP functions, which are able to trigger some buffer overflow vulnerabilities at runtime. To do so, we proposed some lightweight static checks, that can be applied at the binary level. To properly evaluate this work we need to address the following points:

1) The proposed approach should detect functions containing *well-known* buffer overflow vulnerabilities, like the ones of `strcpy` family (`memcpy`, `wcscpy`, etc.).
2) It should also detect *less straightforward* (but still relevant) dangerous functions, for instance functions that have been reported to have (BoF like) vulnerabilities in the past, e.g., those from CVE databases as such. Actually, this is one of our main points: by detecting some known vulnerabilities, we want to convey that our solution will be able to detect *unknown* vulnerable functions as well in the future.
3) It should not classify *all functions* as vulnerable (i.e., the number of false positive should remain weak);
4) Execution cost should remain low enough to allow its execution on real-world (large sized) examples.

We first explain how we chose some experimental dataset to evaluate these four points, followed by the results obtained in terms of execution time and vulnerability detection.

It should, however, be noted at this point that our current implementation does not include the taint analysis and therefore, these results do not consider the presence of tainted paths to such functions while classify them as vulnerable i.e. we cannot say if an attacker can influence their execution. Another thing to be noted is that many of such functions copy string in a secure manner (strncpy or newer string functions of gdi32.dll), but we still detect them as vulnerable mainly because our current implementation does not consider the dependence between loop termination condition and buffer's length. Such considerations constitute our future work.

A. Experimental Dataset

In principle, the Boil criterion we proposed can be computed on any x86 executable file, with or without symbol information However, one of the important issues we want to evaluate is the ability of our Boil criterion to retrieve *known* vulnerability (since we think it is a first condition to find unknown vulnerabilities in the future). To do so, we need to retrieve the *name* of the functions classified as vulnerable by our Boil detection technique. This is the only way to check if they correspond to known vulnerable functions. However, most of the time, function names are not present in the binaries, especially COTS softwares that we plan to experiment with. Therefore, it is not possible to establish that we recognise, for example, `strcpy` function because all we will get is an address. As a consequence we need to find a way to have function names in the executable of the application.

For C/C++ library functions (like the ones of the `strcpy` family), this problem is partially solved by the use of IDA Pro since it can recognise such functions thanks to the so-called FLIRT signatures. Moreover, Microsoft libraries are usually shipped with symbol files (using the `.pdb` format). Thus, if an application binary is compiled statically to use such library functions, IDA pro will recognise them. For this reason we included Microsoft libraries like `msvcr80.dll`, `ntdll.dll`, `kernel32.dll` or `gdi32.dll` in our experiment database.

However, we also need to identify known vulnerable functions that are not known library functions. This means that the corresponding binary executable should contain the symbol table and it should be compiled with some debugging information. Unfortunately, this is not the case for most real-life applications. As a consequence, in order to get a usable dataset, we turned ourselves to resources like CVE Mitre [15] and OSVDB [16] to find applications those source code is available and could be compiled with debugging flags on. Following is the list of applications that we choose for experimentation along with the description of the vulnerability.

- `freeType` CVE ID: 2010-2806; Vulnerability: Array index error in FreeType before 2.4.2; vulnerable function: t42_parse_sfnts() in type42/t42parse.c file.
- `OpenSSL` CVE ID: 2007-5135; Vulnerability: off-by-one buffer overflow in OpenSSL 0.9.7l and 0.9.8d; Vulnerable Function: SSL_get_shared_ciphers().
- `mpg123` CVE ID: CAN-2004-0805; Vulnerability: Multiple buffer overflow vulnerabilities in mpg123-0.59r; Vulnerable Function: II_step_one(), II_step_two() in layer2.c.
- `ImageMagick` CVE ID: 2007-4987, 2007-4988; Vulnerability: Off-by-one error and heap-based buffer overflow in ImageMagick before 6.3.5-9; Vulnerable Function: ReadBlobString() function in blob.c and ReadDIBImage() function dib.c file.
- `TinTin++` CVE ID: 2008-0671; Vulnerability: Stack-based buffer overflow in TinTin++ 1.97.9 and WinTin++ 1.97.9; Vulnerable Function: add_line_buffer() in buffer.c file.

B. Experimental Results

Based on the criteria mentioned in the previous section, we choose the above mentioned open-source applications, and compile them using GCC and Microsoft CL compiler, as is applicable. We believe that these two compilers represent a majority of C/C++ executables that are commonly used. The experiments are executed on a Windows 2003 server machine, with Intel 2 GHz CPU and 4 GB of RAM.

The table I provides details of the experimental results obtained with two versions of our BOIL detection criterion: the more conservative one, described in section II and called hereafter BOIL1, and the extension described in section IV, called BOIL2. Table I gathers the following information:

- the full name of the each module we analyze;
- the total number of functions this module contains;
- the total numbers of loops (as detected by our loop detection algorithm);
- the total time (in seconds) to compute BOP functions in the whole module (we have mentioned the time for BOIL1 only, as the time for BOIL2 is not significantly different)
- the results produced by each BOIL detection criterion (BOIL1 and BOIL2), namely:
  - the number of BOIL loop detected (and their percentage w.r.t the total number of loops);
  - the number of BOP functions detected (and their percentage w.r.t the total number of functions);
  - the vulnerability detected: a **(Y)** in this column indicates that a (known) vulnerable function has been detected in this application (together with its name), whereas a **(N)** indicates that it has been missed by our analysis.

As can be seen in the table I, all known vulnerabilities are retrieved by the BOIL1 analysis (including the known strcpy like functions), while keeping a low rate of false positives (the percentage of BOP functions remains most of the time below 10%). As expected, BOIL2 reduces even more the number of BOP functions detected, without missing real vulnerabilities in most cases. A notable exception occurs for the `freeType` application, where BOIL2 could not identify the vulnerable function `t42_parse_sfnts()`. This is because in the loop responsible for the buffer overflow, the number of iteration depends on the *destination buffer* and, as a result, a false negative is produced by BOIL2.

VI. Related Work

The objective of this paper is to provide a lightweight static analysis technique for vulnerable pattern detection in binary code. However, numerous tools have already been developed for software vulnerability detection. First, we discuss how our work falls in this context, and then we compare it with other static analysis techniques proposed for binary code. Finally, we mention other related work regarding vulnerability pattern detection.

A. Vulnerability detection

Automated vulnerability detection is an important issue in computer security, and numerous research papers and tools have been produced on this topic. Most of the techniques proposed so far can be gathered behind the concept of "smart fuzzing", which consists in detecting potential vulnerabilities at runtime, by executing the target application on well-chosen inputs. Two main research projects illustrate this approach, but many other tools are built on similar ideas like [17]:

TABLE I
LIST OF THE PROGRAMS USED AND CORRESPONDING RESULTS

Module	Total func	Total loops	Total Time	BOIL1			BOIL2		
				BOILs	#BOP func	BOP func names	BOILs	#BOP func	BOP func names
msvcr80.dll	2321	1154	1311.01	188 (16)	113 (4)	*_s versions of strcpy and wcscpy family	153 (13)	103 (4)	*_s versions of strcpy and wcscpy family
GDI32.dll	1775	655	633.81	70 (10)	51 (2)	StringCchCopyA, StringCchCopyNW, ...	39 (5)	33 (1)	StringCchCopyA, StringCchCopyNW, ...
freeType	1910	2568	1948.80	409 (15)	249 (13)	**(Y)** t42_parse_sfnts(), ...	71 (2)	59 (3)	**(N)** ...
OpenSSL/ ssleay32.dll	901	112	181.24	45 (40)	30 (3)	**(Y)** SSL_get_shared_ciphers(), ...	13 (11)	12 (1)	**(Y)** SSL_get_shared_ciphers(), ...
mpg123/ layer2.o	11	23	39.24	7 (30)	2 (18)	**(Y)** II_step_one(), II_step_two()	7 (30)	2 (8)	**(Y)** II_step_one(), II_step_two()
ImageMagick/ blob.o	136	15	14.96	1 (6)	1 (< 1)	**(Y)** ReadBlobString()	1 (6)	1 (< 1)	**(Y)** ReadBlobString()
ImageMagick/ dib.o	41	35	204.81	15 (42)	2 (4)	**(Y)** ReadDIBImage(), ...	14 (42)	2 (4)	**(Y)** ReadDIBImage(), ...
TinTin++/ buffer.o	48	22	15.11	6 (27)	4 (8)	**(Y)** add_line_buffer(), ...	6 (27)	4 (8)	**(Y)** add_line_buffer(), ...
FreeFloat FTP	309	146	54.79	21 (14)	12 (3)	**(Y)** strcpy, wcscpy responsible for BoF. OSVDB ID: 6962121	18 (12)	10 (3)	**(Y)** strcpy, wcscpy responsible for BoF. OSVDB ID: 69621

- Bitblaze [4], is a platform which integrates three main components: a static analysis component, and providing several general-purpose analysis (value-set analysis, weakest precondition, slicing); a dynamic taint-analysis component; and a dynamic symbolic execution engine.
- Sage [5], is routinely used in Microsoft to find software vulnerabilities in Windows applications. Like Bitblaze, It is also based on dynamic symbolic execution and it implements an efficient search space algorithm whose objective is to maximise code coverage.

Although implementing very efficient program instrumentation and path exploration techniques, these two platforms are based on rather simple vulnerability definitions. Vulnerability detection in Bitblaze is guided either by differences between patched and un-patched versions of a given application [18], or by calls to specific dangerous library functions [19]. Sage is essentially a (general-purpose) bug detection tool, able to detect any memory access violation or unexpected memory consumption.

The work we present in this paper is supposed to be used as the first stage of a complete vulnerability detection tool chain, by identifying dangerous code patterns, *before* running any (costly) path exploration technique. As such, it cannot be directly compared with general vulnerability detection tools. However, we believe that it could help to greatly improve the efficiency of such tools, by guiding the search towards the parts of the code that are more likely to contain potential vulnerabilities.

B. Binary level static analysis

Performing static analysis on binary code is challenging due to the lack of information usually available at the source level (e.g, type annotations). Such analysis has been first proposed either for alias detection [20], or approximated value computation [9]. More recently, specific analysis have been developed for security purposes, like taint analysis [21], or string analysis [22].

The analysis we use in our work is simpler, but *cheaper*, than the ones mentioned above. In fact, our initial purpose was to provide a simple criterion, going beyond a purely syntactic check, but still computable on large pieces of code, to be used as a preliminary filter for identifying dangerous functions. That is why this criterion relies only on a combination between loop detection and dependency analysis, defined at the intra-procedural level.

However, it is clear that this criterion could be made more precise (i.e., avoiding more false positives) by adding, for instance, inter-procedural taint propagation, or (some limited) value analysis. More experiments are required to evaluate the potential benefits (in terms of false positive reduction) of such extensions with respect to the cost overhead.

C. Vulnerability pattern detection

To the best of our knowledge, there are not so many works dedicated to vulnerable function computations by mean of static binary analysis. We briefly present below some of them that could influence our future work.

The work by Li and Shieh [23] has a non-empty intersection with our approach. However, there are significant differences. Their tool, RELEASE, is a concolic execution based technique for generating inputs, whereas our approach is based on a pure static analysis technique.

The work by Ketterlin and Clauss proposes a method of reducing instrumentation sites in binary tracing by introducing the idea of *program skeletonization* [24]. This is quite a different objective. However, we could see some overlapping with our work, i.e., induction variable resolution. In our framework, rather than detecting induction variable, we address the loop iteration dependency indirectly by searching for the code construct corresponding to loop termination condition. In [24], this check is formalised on an SSA intermediate representation to explicitly identify induction variable.

Dispatcher (Caballero *et. al.* [25]) is a protocol format reverse engineering framework. As an intermediate step, the method makes use of *dependency chains* which is similar to our method except that the dependency chain tracks *memory writes* backward. In our case, we do not track the values of the memory but the registers only. Another significant difference is that the analysis is done on the execution trace of the application, i.e., it is a dynamic analysis approach.

VII. Conclusions and Future work

Developing automated vulnerability detection techniques, operating at the binary level, and able to address real-word applications is an important and difficult challenge. One of the pre-requisite for such methods is to identify some relevant vulnerability patterns, allowing to restrict the huge search space, and to concentrate the efforts on the most critical parts of the code. Regarding buffer overflows on the execution stack, which is still a commonly exploited vulnerability, many detection techniques focus on the use of some library functions known to be unsafe. The purpose of this paper is to show that with a small time overhead, lightweight static analysis techniques are able to easily detect similar functions, based on simple behavioural patterns, and presenting the same weaknesses as those by library functions from security point of view.

To do so, we propose a detection method based on the computation of self-dependency chains inside loop bodies, leading to the notion of BOILs and BOP functions. After defining BOILs from vulnerability standpoint, we discuss algorithm to detect such loops in executables of the application. We also discuss its implementation. Experimental results are quite encouraging. The present implementation is able to analyse real-world applications. On the data set we used, we showed that vulnerability search could focus on only 10% to 20% of total functions present in the application, while still detecting vulnerable functions. These results also indicate that our proposed method does not make high false positives.

As a future work, we intend to enhance the definition of BOILs by adding other criteria. A very important one is the computation of the taint information flow, to identify BOILs where the source contents are controlled by the user (which makes this iteration highly critical), which should further reduce the number of BOP functions. Other issue would be to deal with more complex loop patterns (memory writes across nested loops, or across several procedures), but we would still need to keep a trade-off between the accuracy of the analysis and the computational cost (since we only want here to identify potential vulnerabilities as a preliminary step, to be completed by other run-time techniques).

Finally, an important perspective is to look for similar "semantic-based patterns" associated to other kinds of insecure programming flaws, like badly controlled uses of the dynamically allocated memory (e.g., heap overflows, or use-after-free, etc.). Detecting statically such patterns would improve current vulnerability detection techniques.

Acknowledgements: The authors would like to thank the anonymous referees for their valuable comments. This work is also the result of fruitful discussions with Marie-Laure Potet.

References

[1] Mitre-Sans, "2011 cwe/sans top 25 most dangerous software errors," http://cwe.mitre.org/top25/.

[2] Grammatech, "Codesurfer," www.grammatech.com.

[3] B. Scholz, C. Zhang, and C. Cifuentes, "User-input dependence analysis via graph reachability," in *IEEE Int. Workshop on Source Code Analysis and Manipulation*, Los Alamitos, CA, USA, 2008, pp. 25–34.

[4] D. Song, D. Brumley, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *In Proc. of the 4th Int. Conf. on Information Systems Security*, 2008.

[5] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[6] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proc. of the 2007 Int. Symp. on Software Testing and Analysis*. New York, NY, USA: ACM, 2007, pp. 196–206.

[7] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *Proc. of the symp. NDSS 02*. The Internet Society, 2000, pp. 3–17.

[8] H. Flake, "More fun with graphs," Technical Talk, BlackHat 2003, 2003, www.blackhat.com/presentations/bh-federal-03/bh-fed-03-halvar.pdf.

[9] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Transactions on Programming Languages and Systems*, vol. 32, pp. 23:1–23:84, August 2010.

[10] N. Mitchell, L. Carter, and J. Ferrante, "A modal model of memory," in *Proc. of the Int. Conf. on Computational Sciences-Part I*. London, UK: Springer-Verlag, 2001, pp. 81–96.

[11] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[12] Zynamics, "Binnavi- binary code reverse engineering tool," http://www.zynamics.com/binnavi.html.

[13] Hex-Rays, "Ida pro disassembler and debugger," http://www.hex-rays.com/products/ida/index.shtml.

[14] K. Kratkiewicz and R. Lippmann, "A taxonomy of buffer overflows for evaluating static and dynamic software testing tools," in *Proc. of Workshop on Software Security Assurance Tools, Techniques, and Metrics*, ser. NIST Special Publication 500-265. US: NIST, 2005, pp. 44–51.

[15] http://cve.mitre.org/.

[16] http://osvdb.org.

[17] M. Cova, V. Felmetsger, G. Banks, and G. Vigna, "Static detection of vulnerabilities in x86 executables," in *Proc. of the 22nd Annual Computer Security Applications Conference*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 269–278.

[18] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proc. of the 2008 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 143–157.

[19] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. of NDSS 2005, San Diego, California, USA*. The Internet Society, 2005.

[20] S. Debray and R. Muth, "Alias analysis of executable code," in *In POPL*, 1998, pp. 12–24.

[21] S. Cheng, J. Yang, J. Wang, J. Wang, and F. Jiang, "Loongchecker: Practical summary-based semi-simulation to detect vulnerability in binary code," in *Proc. 10th Int. Conf. on Trust Security and Privacy in Computing and Communications*. IEEE, 2011, pp. 150–159.

[22] M. Christodorescu, N. Kidd, and W.-H. Goh, "String analysis for x86 binaries," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 1, pp. 88–95, 2005.

[23] B.-H. Li and S. Shieh, "Release: Generating exploits using loop-aware concolic execution," in *Proc. of the 2011 Fifth Int. Conf. on Secure Software Integration and Reliability Improvement (SSIRI)*, june 2011, pp. 165 –173.

[24] A. Ketterlin and P. Clauss, "Efficient memory tracing by program skeletonization," in *Proc. of 2011 IEEE Int. Symp. on Performance Analysis of Systems and Software (ISPASS)*, april 2011, pp. 97 –106.

[25] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering," in *Proc. of the 16th ACM conference CCS '09*. New York, NY, USA: ACM, 2009, pp. 621–634.