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.

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
moves the corresponding characters to p (i.e., to the destination). Functionally 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 BOPs and BOILs

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 Buffer Overflow Inducing Loops (BOIL). We call such a function Buffer Overflow Prone (BOP) function. A loop is a BOIL if there is memory write within the loop and that memory is changing within the loop.

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 compiled to one common byte-code (e.g. .NETish 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 \( X[c] = \ldots \).
- **Stride access**: memory locations are accessed with a fixed stride, representing the distance between two successive write addresses, as in \( X[f(i)] = \ldots \).

\(^1\)http://msdn.microsoft.com/en-us/library/8db701c(v=vs.80).aspx
Listing 2. Assembly code of the strcpy function

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 $X[f(y)] = \ldots$

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 ($*p++ = \ldots$). 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 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:

\[
MEM_{cur} = MEM_{pre} + \text{stride}
\]

where, \(MEM_{cur}\) is the current memory location and \(MEM_{pre}\) is the previous memory location, and \(\text{stride}\) 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
  \[
  \text{adr} = \text{adr} + \text{stride}
  \]
  \[
  \text{MEM}[\text{adr}] = \ldots
  \]
- Since we focus on stack-based buffer overflow detection we assume that address \(\text{adr}\) is either contained in a register, or it results from some combination of the base register \(\text{ebp}\) and some offset.

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

\[ I = [s_1, s_2, \ldots, s_n] \] is an execution sequence (where each \(s_p\) is an instruction);
\[ K = [s_{k_1}, s_{k_2}, \ldots, s_{k_m}] \] is called a sub-sequence of \(I\) if it contains some of the instructions of \(I\) taken in this order, namely \((i)\) \(\forall k \in [1, m]. s_k \in I\) and \((ii)\) \(k_i < k_{i+1}\);
\(v\) denotes an arbitrary memory location (that could be either a register, 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 \(I\) is a sequence of memory locations \(v_1, v_2, \ldots, v_m = v\) such that \(s_{k_1}, s_{k_2}, \ldots, s_{k_m}\) is the longest sub-sequence of \(I\) 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 \(I\) is a backward dependency chain \(v_1, v_2, \ldots, v_m\) s.t. \(v_m = v_i\) for some \(i \in (1, 2, \ldots, m - 1)\).

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 \(v_i = v_j, j \in (2, 3, \ldots, i - 1)\) (Fig. 1(ii))

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 \(\text{edi}\) 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):

\[
\begin{align*}
1: & \ \text{MEM}[\text{edi}] \leftarrow \text{SRC} \\
2: & \ \text{edi} \leftarrow \text{edi} + 1 \\
3: & \ \text{MEM}[\text{edi}] \leftarrow \text{SRC}
\end{align*}
\]

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):

\[
\begin{align*}
1: & \ \text{reg} \leftarrow \text{MEM}[\text{base}+\text{offset}]
\end{align*}
\]

Fig. 1. Self-dependency chain graph shape.
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:

c) **Extended Self-Dependency Chain:** There exists an extended self-dependency chain from a memory location \( v \) over an execution sequence \( I \) if there exists a non-empty set of dependency chains \( C_1 = [v_1^1, \ldots, v_n^1], C_2 = [v_1^2, \ldots, v_{n_2}^2], \ldots, C_p = [v_1^p, \ldots, v_{n_p}^p] \) such that:
- \( \forall i \in [1, p - 1], \ v_{n_i}^i = v_{i+1}^i \);
- \( v_{n_p}^p = v \);
- \( C_1 \) is a self-dependency chain.

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

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 \( L \) 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 \( L^* \), \( v \) 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.

c) **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: \( \text{adr} = \text{adr} + 0 \)
2: \( \text{MEM}[\text{adr}] = \ldots \)

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.
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 

\[ \text{stm src, dest; where src is a literal or register whose value is stored at the memory address contained in register dest.} \]

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 + 0xFFFFFFFF, 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:

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 BOILs

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].
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

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 0.9.7i 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 `blos.c` and ReadDIBImage() function in `dib.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 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]:
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 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

<table>
<thead>
<tr>
<th>Module</th>
<th>Total func</th>
<th>Total loops</th>
<th>Total Time</th>
<th>BOIL1</th>
<th>BOIL2</th>
<th>BOIL3</th>
<th>BOIL4</th>
<th>BOIL5</th>
<th>BOIL6</th>
</tr>
</thead>
<tbody>
<tr>
<td>msvcr80.dll</td>
<td>2321</td>
<td>1554</td>
<td>1311.01</td>
<td>188 (16)</td>
<td>113 (4)</td>
<td>153 (13)</td>
<td>153 (13)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GD32.dll</td>
<td>1775</td>
<td>665</td>
<td>633.81</td>
<td>70 (10)</td>
<td>51 (2)</td>
<td>39 (5)</td>
<td>33 (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>freeType.dll</td>
<td>910</td>
<td>112</td>
<td>181.24</td>
<td>45 (40)</td>
<td>30 (3)</td>
<td>13 (11)</td>
<td>12 (1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenSSL/</td>
<td>1910</td>
<td>2568</td>
<td>1948.80</td>
<td>409 (15)</td>
<td>249 (13)</td>
<td>71 (2)</td>
<td>59 (3)</td>
<td>(N)</td>
<td></td>
</tr>
<tr>
<td>ssl32.dll</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(Y)</td>
<td>(Y)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mpg123/layer2.o</td>
<td>11</td>
<td>23</td>
<td>39.24</td>
<td>7 (30)</td>
<td>2 (18)</td>
<td>7 (30)</td>
<td>2 (8)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ImageMagick/</td>
<td>136</td>
<td>15</td>
<td>14.96</td>
<td>1 (6)</td>
<td>1 (&lt; 1)</td>
<td>1 (6)</td>
<td>1 (&lt; 1)</td>
<td>(Y)</td>
<td>(Y)</td>
</tr>
<tr>
<td>blob.o</td>
<td>41</td>
<td>35</td>
<td>204.81</td>
<td>15 (42)</td>
<td>2 (4)</td>
<td>14 (42)</td>
<td>2 (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ImageMagick/</td>
<td>48</td>
<td>22</td>
<td>15.11</td>
<td>6 (27)</td>
<td>4 (8)</td>
<td>6 (27)</td>
<td>4 (8)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dll.o</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(Y)</td>
<td>(Y)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tiff++/</td>
<td>309</td>
<td>146</td>
<td>54.79</td>
<td>21 (14)</td>
<td>12 (3)</td>
<td>18 (12)</td>
<td>10 (3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>buffer.o</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**TABLE I**

**LIST OF THE PROGRAMS USED AND CORRESPONDING RESULTS**
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-requisites 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