International Journal of Applied Engineering Research ISSN 0973-4562 Volume 10, Number 14 (2015) pp 35006-35022

Information Flow Tracking Approach for Vulnerability Signature Generation in Web Applications

Raman Dugyala
Dept. of Computer Science, Vardhaman College of Engineering, Hyderabad, Telangana, India
E-mail: raman.vsd@gmail.com

Bruhadeshwar Bezawada
Independent Researcher, Hyderabad, Telangana, India.
E-mail: bezawada@iiit.ac.in

Rajini Kanth V. Thatiparthi
Dept. of Computer Science, Sreenidhi Institute of Science and Technology, Hyderabad, Telangana, India
E-mail: rajinitv@gmail.com

Sai Sathyanarayan
Independent Researcher, Singapore.
E-mail: sai.sathyanarayan@gmail.com

Abstract: Many existing web browser software suffer from vulnerabilities that can be exploited by attackers to compromise critical applications. Such vulnerabilities pose a significant threat to the millions of important web-based transactions executed by users across the globe. The challenge in this domain is to detect such exploits before they are accepted by a vulnerability condition inside the software. Several approaches have been proposed, such as vulnerability based signatures, regular expression based signatures, blacklisting inputs and user input validation. Among these approaches, vulnerability based signatures use the semantics of the vulnerability to create a signature that is not only sound and accurate, but also has a faster exploit detection rate than pattern based signature generation techniques. However, existing approaches for generating vulnerability based signatures analyze all possible program execution paths leading to the vulnerability point and generate the signature. Clearly, the complexity of such an exploration is quite high due to the large number of program paths and the state-of-the-art approach has a $O(N^2)$ complexity, where N is the number of execution paths leading to the vulnerability point. In this work, we focus on the problem of generating vulnerability based signatures for web application software in an efficient and scalable manner. We propose a novel linear time $O(N)$ approach for automatic creation of vulnerability based signatures, which recognizes different exploit variants of the vulnerability, regardless of the execution paths through which they propagate in the program. By linear time, we mean that the path exploration done by our approach is $O(N)$ complexity and the signature generated is much smaller than the actual program itself. Our algorithm explores the program execution in a "backward" direction with the vulnerability as the starting point. Our approach can be used on programs where source code is available, on binary executables and on shared libraries, regardless of the size of the program under consideration. We have created a number of vulnerability based signatures for widely used open source web-application software, such as ATPhttpd and Ghttpd. Our experimental results

on signature matching show that our signatures are able to match exploit inputs at a rate of 1.5 to 7 times faster than existing approaches. Furthermore, our approach can be easily extended to generate a single aggregated signature for multiple vulnerabilities.

AMS subject classification:
Keywords: Information Flow Analysis, Vulnerability Signature Generation, Behavior Tracing, Commercial Software, Exploits.

1. Introduction

A vulnerability is a state in a software which causes unintended or malicious behavior when executing specially crafted inputs which are called exploits. Software vulnerabilities are a major threat to many applications. The computer emergency response team (CERT) and application vendors report a number of vulnerabilities almost on a daily basis. A serious cause for concern is that attackers are able to generate exploits at a faster rate than the rate of patch generation for vulnerabilities. For example, the average patch release time by Microsoft [1] is around 70 days. This situation arises due to the inherent difficulty of analyzing vulnerabilities, the time taken for generating patches and performing regression testing of the patched application. The situation is exacerbated as most of the vulnerabilities share common features and therefore exploit writers do not have to exert themselves to generate new exploits. The time lag between vulnerability report and patch generation indicates that, though patching remains a long term solution, application vendors need to examine alternate approaches to protect their applications. Thus, the protection of applications from software vulnerabilities on a real-time basis is a critical problem.

One popular approach to protect against vulnerabilities is to use signature based filtering to filter out inputs that exploit the vulnerabilities. The challenge in this solution is to generate a signature that can precisely characterize all possible exploit inputs. In network intrusion detection systems [2–12] the signature is represented as a pattern which is extracted from known exploit inputs. These approaches rely on a few sample exploits, that are typically published by attackers, to characterize the exploit inputs. However, the limitation in these approaches is that, the set of exploits that is detected is limited to the patterns extracted from the training samples. If there are exploits that are unknown and are not used during the training phase then such exploits cannot be detected. Recent research has shown that an attacker can often fool these approaches into generating highly inaccurate signatures [13, 14]. The main challenge in these approaches is that the signature generation process does not consider the semantics of the actual vulnerability and therefore, may not cover the entire exploit space completely. Hence, in order to achieve comprehensive coverage of the input exploit space, it is important to consider the semantics of the vulnerability when generating the signature.

1.1. Drawbacks of Prior Work

Vulnerability-based signature generation has been described in [15], [16], [17]. The main feature of these approaches is to automatically generate a signature based on the vulnerability itself. Identification of the vulnerability point can done through known approaches [18], [19] or by running a sample exploit. In these approaches, given the vulnerability point, the vulnerability is characterized by analyzing all possible program execution paths that reach the vulnerable statement from the entry point of the program. To facilitate this process, the control flow graph based on the intermediate representation (IR) of the binary program is generated and the reachability analysis is performed on this graph. The final signature is essentially a symbolic program that covers the execution paths taken by the vulnerability . The authors [15], [16], [17] show that it is possible to convert this symbolic signature into a finite automaton or a regular expression. However, these approaches have two limitations. First, the complexity of analysis is quite high, i.e., $O(N^2)$ [16], where N corresponds to the number of lines of code or executable paths in the program. One reason for the complexity of the current approaches is that, exploration of the state space in a forward direction is difficult task. For example, a single loop when unrolled can generate possibly an exponential number of states [15]. Even with optimizations such as using weakest pre-conditions [16], the worst case state exploration is still $O(N^2)$.

For programs with more than 1 million lines of code (e.g., Opera web browser [20]) the signature generation can be expensive. Second, a more important shortcoming, their approach generates the signature using the entry point of the program. Such forward exploration needs to factor indirect jumps, which can only be determined at run-time. Also, if the starting point of the state exploration changes then the entire process needs to be repeated. Many vulnerabilities have been reported in shared libraries such as Microsoft Dlls. Thus, we make the observation that, though forward state exploration [15], [16], [17] is a sound technique, the complexity is quite high. Hence, an exploit may enter the program through different points in the shared library, which implies that there is a need for a separate signature that corresponds to each entry

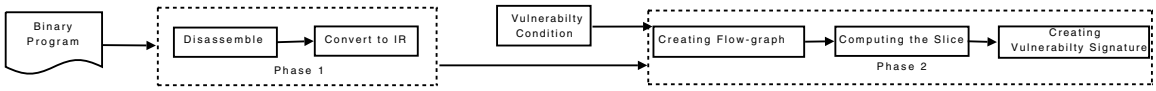

FIG. 1. An architecture to compute a vulnerability signature.

point and thereby, increasing the size of the signature and the complexity of signature generation.

1.2. Proposed Approach

In this work, we describe an approach for the efficient and automatic generation of vulnerability based signatures that addresses the shortcomings identified in existing approaches [15], [16], [17]. One reason for the complexity of the current approaches is that, exploration of the state space in a forward direction is difficult task. For example, a single loop when unrolled can generate possibly an exponential number of states [15]. Even with optimizations such as using weakest pre-conditions [16], the worst case state exploration is still $O(N^2)$. The second reason is that forward exploration needs to factor indirect jumps, which can only be determined at run-time. Also, as stated earlier, if the starting point of the state exploration changes then the entire process needs to be repeated. Thus, we make the observation that, though forward state exploration [15], [16], [17] is a sound technique, the complexity is quite high. Therefore, to address this issue, we consider the exploration of the states starting from the vulnerability point to the beginning of the program. This allows us to prune execution paths that do not lead to the vulnerability point thereby, making our signature generation efficient as well as accurate.

Our algorithm is a simple variation of the traditional slicing technique [21]. The input to our approach can be the source of a program, a binary executable or a shared library. We use the IDApro [22] disassembler to generate the intermediate representation (IR) of the code and augment it with labels using customized plugins. We perform a static analysis on the resulting IR code to generate the vulnerability signature. Initially, we identify the variables involved in the vulnerability condition i.e., the variables that cause the condition to be satisfied. Now, starting from the vulnerability condition we trace backwards, marking those statements that affect the vulnerability variables directly or indirectly. By doing so we are able to guarantee that all paths that lead to the vulnerability condition are covered even in the presence of direct or indirect jumps. Furthermore, we use the static ordering of statements to resolve jumps, loops and conditional statements. The presence of procedures or sub-routines is not an issue since we trace the execution of the variables within the function and the relevant statements are marked as well. The vulnerability based signature is then the set of statements that are output by the tracing algorithm. The generated symbolic signature is converted into a 'C' program or can be converted into a regular expression as in [15]. The complexity of our algorithm is linear $O(N)$ in the number of statements contained in the program. Our architecture is shown in Figure 1.

Signatures generated by our approach are sound as all possible paths of vulnerability exploitation, including those that result from sub-routines or helper functions, are covered. Our signature when compiled as a 'C' program and executed returns "EXPLOIT" for all exploit inputs and "BENIGN" otherwise. As with earlier approaches, we have dealt with the problem of non-terminating loops by bounded iteration [16], [23], [17], [15] as solving the general problem of loop halting is undecidable. Our experimental results have, so far, not yielded a false negative due to this feature.

1.3. Key Contributions

- We describe an automatic vulnerability-based signature generation algorithm which deals effectively with the state exploration problem. The run-time of our algorithm is $O(N)$ in the worst-case where N is the number of program statements. The size of our signatures is much smaller compared to the original program. To show the efficiency of our approach, we have chosen various programs having a few thousand lines, like ATPhttpd, to large programs, like the Opera web browser, having more than a million lines. In earlier approaches like [15], [16], [17], analyzing such large programs is an expensive task and hitherto, such attempts have not been made. The scalability of our approach in handling such large programs is a major contribution of this work. Also, due to nature of the backward tracing, our approach can handle multiple vulnerabilities enabling us to create a single input filter that can effectively filter out exploit inputs targeted to any of different vulnerabilities in the program.

- We have implemented a prototype system and have successfully created signatures for a whole range of programs. In addition to generating signatures for binary executables, our approach is the first approach that can generate vulnerability based signatures for dynamically shared libraries like Microsoft DLLs which have a number of reported vulnerabilities. This result is significant as our approach can protect a whole range of applications that use such shared libraries. Specifically, we have picked the most commonly used mshtml.dll, netapi32.dll and created signatures for the vulnerabilities present in them. However, due to inherent complexity by which DLLs are used by applications we have not been able to test them fully.
- Our performance evaluation shows that, our signatures can detect exploit inputs at a much faster rate than existing approaches [17] of the order of 1.5 to 7 times speed in exploit detection rate. The rate for matching benign inputs is even faster. We have trial run our signatures for over a million inputs and reported the results. These results show that our signatures are amenable to be deployed in high-speed applications which process thousands of inputs per second.

Organization. The paper is organized as follows. In Section 2, we describe the problem statement, notation and outline related work in the area. In Section 3, we describe our signature generation approach. In Section 4, we show our results and discuss the various issues involved. We conclude the paper in Section 5.

2. Problem Statement

In this section we introduce our terminology for vulnerabilities and vulnerability signatures. Intuitively, a vulnerability signature is representation for the set of inputs that satisfy a specified vulnerability condition. The problem of vulnerability based signature generation is stated as follows:

Problem Statement. Given a program P, we assume that, a site has detected a vulnerability in the program through some means such as dynamic taint analysis [19], [24]. The problem is to create a symbolic signature by statically analyzing the program P and exploring all possible program paths that can reach the vulnerability condition. It is desired the signature generation be as quick as possible and that the generated signature comprehensively filter all possible exploit inputs.

2.1. Preliminaries

In Table 1, we show the notation used in our paper. We define a set T, which contains the "operation" on a variable associated with a program state v in a program P (e.g., Define, Use, Modify etc.). Therefore T would be of the form,

$$T = \{Define, Use, Set, Modify, DerefUse, DerefSet, Literal\}$$

In the set T, for at a program state v, $Define$ means a variable is declared at v, Set means that the variables are assigned some value at v, and Use means that some variables are being

Table 1. Description of Notations

Notation	Definition	Notation	Definition
P	P is the binary program	V_c	Vulnerability Condition of a program P
S	Vulnerability Signature of P	V_p	Vulnerability Point of a program P
V_v	Vulnerability variable of a program P	v	program state in the P
V_0	the entry state of the input to P	T	is a set, which contains the "operation" on a variable.
V_{left}	contains the variable names at the left side of the assignment statement at state v in P	V_{right}	contains the variable names at the right side of the assignment statement at state v in P
$match(V_{left}, T)$	outputs a variable from set V_{left}, where the corresponding variable "type" matches with T	$match(V_{right}, T)$	outputs a variable at set V_{right}, where the corresponding variable "type" matches with T

referenced during the computation associated with the state v. The labels, *Defref Use and Set* are similar to Set and Use but these are operations on dereferenced variables, $Literal$ means that the variable is being assigned to a constant at v and, finally, $Modify$ means that the variables are being modified at v.

Vulnerability Specification. The *vulnerability specification* (V_p, V_c) is a specification of vulnerability [15], [16], [17] consisting of a *vulnerability point* V_p where the vulnerability program P may "go wrong", and *vulnerability condition* V_c that specifies what is "going wrong". Intuitively, the vulnerability point is the first instruction which may cause unsafe execution. We say that a program with vulnerability specification (V_p, V_c) is exploited when it reaches V_p in a state that satisfies the predicate V_c.

Vulnerability Signature. A vulnerability signature $S_{<P,V_p,V_v>}$ is an input filter for the program P that recognizes exploits of the vulnerability (without needing to run the original program P). Abstractly, the vulnerability signature is a program $S_{<P,V_p,V_v>}$ which takes as input any input x from the program input domain I_D, and returns either EXPLOIT or BENIGN. In our approach, the vulnerability signature is represented as a C program. However, it can be transformed into other representations such as those described in [15], [2], [16].

2.2. Related Work

The protection of software applications has been an active area of interest due to its importance. Here, we give an overview of various approaches to this problem. We have classified these approaches as: signature generation approaches, and memory protection approaches.

Signature Generation Approaches. Exploit based signature generation approaches have been described by many researchers [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]. These approaches take several examples of an exploit, and extract common patterns to use as a signature. We call these *exploit-based signatures*. The main shortcoming of these exploit-based signature is that they are based on specific exploit instances and may have both false-positive and negatives. And also, exploit-based signatures may be defeated with polymorphic worms [25],[7]. Sheild [26] provides vulnerability-specific exploit generic protection. However, it uses manually generated signatures. Costa et al. [2] propose a concurrent work to automatically generate *host-based* input filter, which has greater accuracy than network-based input filters, and can correctly recognize some semantically equivalent inputs. However this suffers when the input is in encrypted format.

In this work, we focus on the more accurate vulnerability based signatures. InroVirt [27] uses vulnerability-specific predicates to detect when a vulnerability has been generated. However the predicates are generated manually. Automatic vulnerability based signatures were first described by Brumley *et al.* [17], [15], [16]. In [15], the authors show the possibility of generating vulnerability based signatures by tracing the path of a sample exploit to the vulnerability point. However, their approach required a state exploration which is exponential in the program size. In [16], the authors improve the analysis by using weakest pre-conditions and reducing the complexity of program path generation to $O(N^2)$. In [17], the authors describe an approach to generate signatures even when an exploit can take multiple paths to reach the same vulnerability. The main feature of the approach in [17] is the introduction of an "UNKNOWN" state in the presence of an input which does not match either benign or exploit inputs. The main difficulty in these approaches is the despite the soundness of the signatures they are either too expensive [15], [16] or the assumption of a single entry point into the program [17].

Some approaches like AutoPAG [1] have been proposed to generate patches for software programs. The main goal of this approach is automatically generate patches for a particular vulnerability condition. Their approach is similar to our approach in spirit. However, their approach requires the program source code and also, they rewrite the original program with the generated patch. Further, the complexity of their approach is higher than our approach as they need to perform multiple passes over the program to generate the patch. An important feature that separates our approach from any other proposed approaches is the ability to generate signatures for shared libraries.

Memory Protection Approaches. Several approaches have been proposed to defeat memory corruption attacks. Approaches such as StackGuard [28], StackShield [29], ProPolice [30] and LibSafe [31] and TIED-LibSafePlus [32] try to defend against buffer overflows. Other approaches like FormatGuard [33], FormatShield [34], LibFormat [35], Kimchi [36], Lisbon [37], White-Listing [38] defend against format string attacks. These approaches are limited to certain types of vulnerabilities or exploits, and many of them even require source code of the programs to be protected. Address Space Randomization [39], [40], [41] randomizes the location of various memory segments and other security critical structures, making it hard for an attacker to guess the correct address. Static taint analysis has been used to find bugs in C programs [42], [43], [44] or to find potential sensitive data in Scrash [45]. Binary instrumentation for taint-tracking was first used in TaintCheck [46]. While effective in

attack detection, their approach slows down programs significantly (by about 37x). TaintTrace [47] achieved significantly faster taint-tracking by using more efficient instrumentation based on DynamoRIO. LIFT [48] achieved significant additional performance benefits by using better static analysis and faster instrumentation techniques. However, these approaches require rewriting the binary executable and result in slowdown of the performance. Furthermore, these approaches cause the application to reset upon detection of the attacks. Signature based approaches perform offline processing and do not execute the input on the actual program until it has been determined to be benign. In the next section, we describe our algorithm for generating vulnerability based signatures.

3. Our Approach

In this section, we describe our signature generation approach and illustrate the signature generation on programs where, (a) source code is available and, (b) stripped binary executable is available. The technique for generating signatures for shared libraries, like DLLs, is similar depending on whether the source or binary of the shared library is available. In Section 3.1, we state the signature generation algorithm. In Sections 3.2 and 3.3, we illustrate the signature generation on sample programs.

3.1. Signature Generation

Our algorithm for computing a vulnerability signature for program P, given the vulnerability variable V_v, and vulnerability point V_p is shown in Algorithm 1. The high level idea of this approach is that the signature models the program's vulnerability directly, thus we can guarantee our signature's accuracy with respect to the vulnerability. We start from the vulnerability point and apply the backward slicing algorithm from Figure 1. The generated signature contains all the paths from *vulnerability condition* till a *point*, V_0, in the program P. This point can be the statement where the input, that affects the vulnerability condition, is read from the external source (or) to a particular function (earliest possible function invocation in case of DLL).

Now, given the vulnerability variable(V_v) and the point(V_p), we identify the state of V_v that can satisfy the vulnerability condition e.g., length of V_v that can cause the overflow. Since the value of V_v can be affected in many ways, than just direct user inputs, we need to monitor all such variables that affect this variable. In particular, our algorithm traces out the program statements that contribute to the computation of V_v either, directly by operating on V_v, or indirectly, through assignment or copy from other variables. For instance, if the overflowed variable is a pointer, we find out its declaration statement, its scope and aliases, as well as possible references and dereferences. Since the value of V_v can be affected in many ways other than the direct user inputs, we need to monitor all such variables that affect this variable. Hence, for completeness, we treat all variables that affect V_v as vulnerability variables as well.

Towards this, we define a set "monitored set" (MS), which keeps track of the flow of vulnerability variables in the program. First, we initialize the monitored set with the vulnerability variable V_v from the vulnerability condition V_p. As the program backtracks from V_p, each variable that affects V_v gets added to the monitored set and the statements that contain the any variable from the monitored set get added to the signature. These statements correspond to the *data flow sequence of the signature*. The set of computations that are monitored are given in Table 2. The program state in P also contains 3 sets (*left, right, MS*) apart from the program statement, shown in Table 2, where left set contains variables present in V_{left} (see Table 1), similarly right set.

Note that, while backtracking we also keep track of the looping or branching statements that have an effect on the MS. These statements give the *control flow sequence* for the signature. When our algorithm encounters a looping condition or branching condition, then it adds it to the signature only if the loop has an affect on any vulnerability variable. Note that, the detection of looping or branching conditions can be done on the knowledge of the programming language constructs. Even though the loops are added to the signature, again, only those relevant statements that affect the vulnerability variables are only added to the signature. Once a loop is added to the signature, the variables in the looping condition get added to the MS. In this case, we do a second pass over the loop and add all the statements that include the new vulnerability variables. The reason for this is that the looping condition might be affected due to the vulnerability variables within the loop. In which case, these variables will also need to be added to the MS. On the other hand, for branching statements, this is not the case. We do a single pass over the branch statements and add whatever statements that are discovered. Since the variables in the branching condition cannot be affected by any variable within the branch statements, we do not perform a second pass over the branching statements. Instead, we simply add the variable present in the branching condition in the MS and proceed further.

Optimization. To avoid the complexity of monitoring too many vulnerability variables we examine assignment statements that reset the variable value. For example, consider that a vulnerability variable, say V_m, is set to zero or NULL at

some point in the backtracking. We remove this variable from the MS as whatever operations that may done on this variable prior to this statement have no affect on the statements discovered so far. We are only interested in tracking the flow of this variable after this reset statement. Therefore, the overhead of monitoring this variables is reduced increasing the speed our backward slicing algorithm.

Handling Multiple Vulnerabilities. Note that, the above approach generates a signature which is a subset of the existing program. If a new vulnerability is reported then there is no need to use a separate signature for filtering exploit inputs of that vulnerability. The approach is to generate a signature using our basic approach on the new vulnerability. Now, since both the signatures are generated from the same source code, they can combined in a straightforward manner and all repeated statements can be removed as well. Finally, this combined signature will have two exploit matching conditions, one each for the two vulnerabilities. Thus, instead of deploying a separate signature for each reported vulnerability, we can use a single signature for multiple vulnerabilities and reduce the overhead of exploit detection.

3.2. Signature Generation for Programs with Source Code

In Figure 1(a), we show the source code of a program that contains a buffer overflow vulnerability in line 13. Suppose this vulnerable program is executed with a long string argument (more than 4 bytes) that will overflow the local variable p, our signature will return the value `Exploit`, protecting from buffer overflow. In Figure 1(b), we show the set of states that affect the vulnerability condition which is shown in concentric circles. In Figure 1(c), we show the control flow sequence that connects the data flow sequence. Finally, in Figure 1(d) we show the generated vulnerability signature in 'C' language notation.

Algorithm 1 Algorithm to generate the vulnerability based Signature S

`Let` P be the program, $processed(v)$ keeps track of whether the state in a looping conditional statement is processed or not at state v and, MS_g be the global monitored set.
Initially MS_g set contains the vulnerability variable V_v and S holds the states which contains the vulnerability variable V_v
`Set` $MS_g = \{V_v\}$ and $S = v$
Traversing each program state in backward fashion, we start traversing from the vulnerability point V_p
For each state v in P {
if(v is a looping conditional statement & $processed(v) = 0$) {
$MS_g = MS_g \cup V_{left}$
Pass the control to a state v where the looping conditional statement ends
$processed(v) = 1$; }
elsif(v is a looping conditional statement and $processed(v) = 1$) {
MS(v,P) = V_{left}
$S = S \cup v$
$processed(v) = 0$; }
elsif(v is a assignment statement || conditional statement){
if(($V_{left} \in MS_g$) & (match(V_{Right},"literal")=0)) {
$MS_g = MS_g - V_{left}$
MS(v,P) = $\varnothing$
}
elsif($V_{Left} \in MS_g$) {
MS(v,P) = $MS_g \cup V_{right}$)
$S = S \cup v$
}
else MS(v,F) = $\varnothing$
$MS_g = MS_g \cup$ MS(n,P)
}
Repeat till v cannot be processed further

3.3. Signature Generation for Stripped Binary Executable

Generating signature for a binary executable is similar to source code but as the binary programs are unstructured it is not possible to apply our algorithm directly. To get a structured representation, we convert the binary program into intermediate representation (IR) and apply the algorithm over this IR. We describe the steps involved in the following description. The signature is shown in Figure 4.

Using IDAPro [22], we first disassemble the binary and identify the functions. The IDAPro disassembler does not require the symbol table and can identify the functions using the prologue and epilogue of the executable. In our analysis, we find out if a particular function returns to the caller or not. Using the output of the disassembly we perform preprocessing to extract this information. For example, these functions can some optimizing code inserted by the compiler when the executable was generated. This analysis improves the listing quality because many wrong execution paths are detected and truncated at early stages. Next, we convert the disassembled instructions into an intermediate representation(IR). This intermediate representation is in a C-like language. Figure 2 shows the binary program P and intermediate representation of P. Once the IR is constructed from the

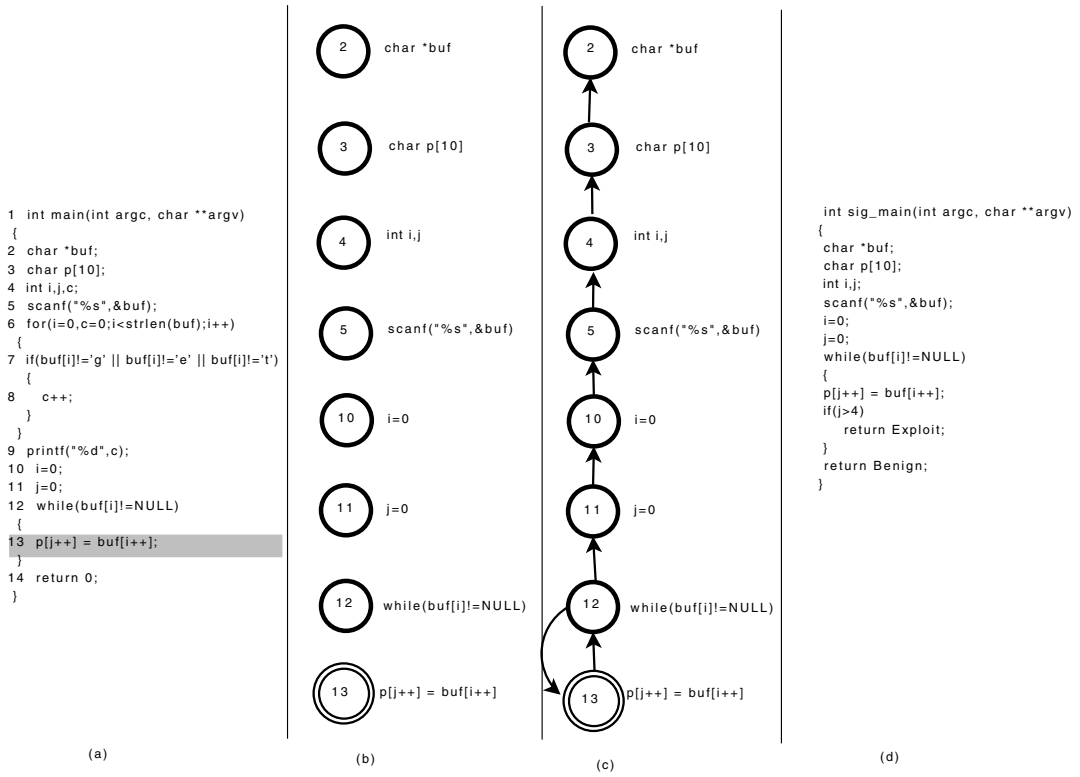

FIG. 2. Signature Generation Process (a) Original Program P, (b) Data Dependencies w.r.t V_p (c) Control Flow Graph and w.r.t V_p, (d) Vulnerability signature for P

Table 2. Monitored set of the program P on each program statement

State #	Left Set	Right Set	Monitored Set
13	p:Deref Set, j:Modify	buf: Deref Use, i:Modify	p, j, buf, i
12[1]	buf:Use	-	buf, p, i,j
11	i:Set	0:Literal	p, j, buf, i
10	j:Set	0:Literal	p, j, buf
9	c:Use, print:Call	-	-
8	c:Modify	-	-
7[1]	buf:Use, i: Use	-	-
6[1]	i:Set, c:Set	strlen:Call, 0:literal	-
5	buf:Init, scanf:Call	-	p, buf

Conditional Statement

previous step, we now apply our signature generation algorithm, Algorithm 1, over the generated IR. These steps are similar to the steps discussed for the source code example from Section 3.2.

3.4. Signature Generation for Shared library

The signature generation for shared libraries is similar to that of the previous two approaches depending on whether we have the source code of the library or the binary version available with us. Due to size of the shared libraries that we have analysed, it is difficult to reproduce the process here. We note that, shared libraries are similar to regular programs except that they do not have a single point of entry. Note that, our backward slicing algorithm traces the vulnerability variables to the maximum possible extent. Thus, regardless of the point of entry into the shared library, the signature will be able to match the exploit inputs. The only difference with respect to the previous two signatures is that the signature for the shared library will need the state of the other variables at the point of entry into the program. However, note that, the variables of the library cannot be affected by concurrent executions i.e., the state of the variables is decided only after an application enters into the shared library. Therefore, we can initialize the signature with default values for each possible entry point in the shared library and catch the exploit input regardless of the entry point.

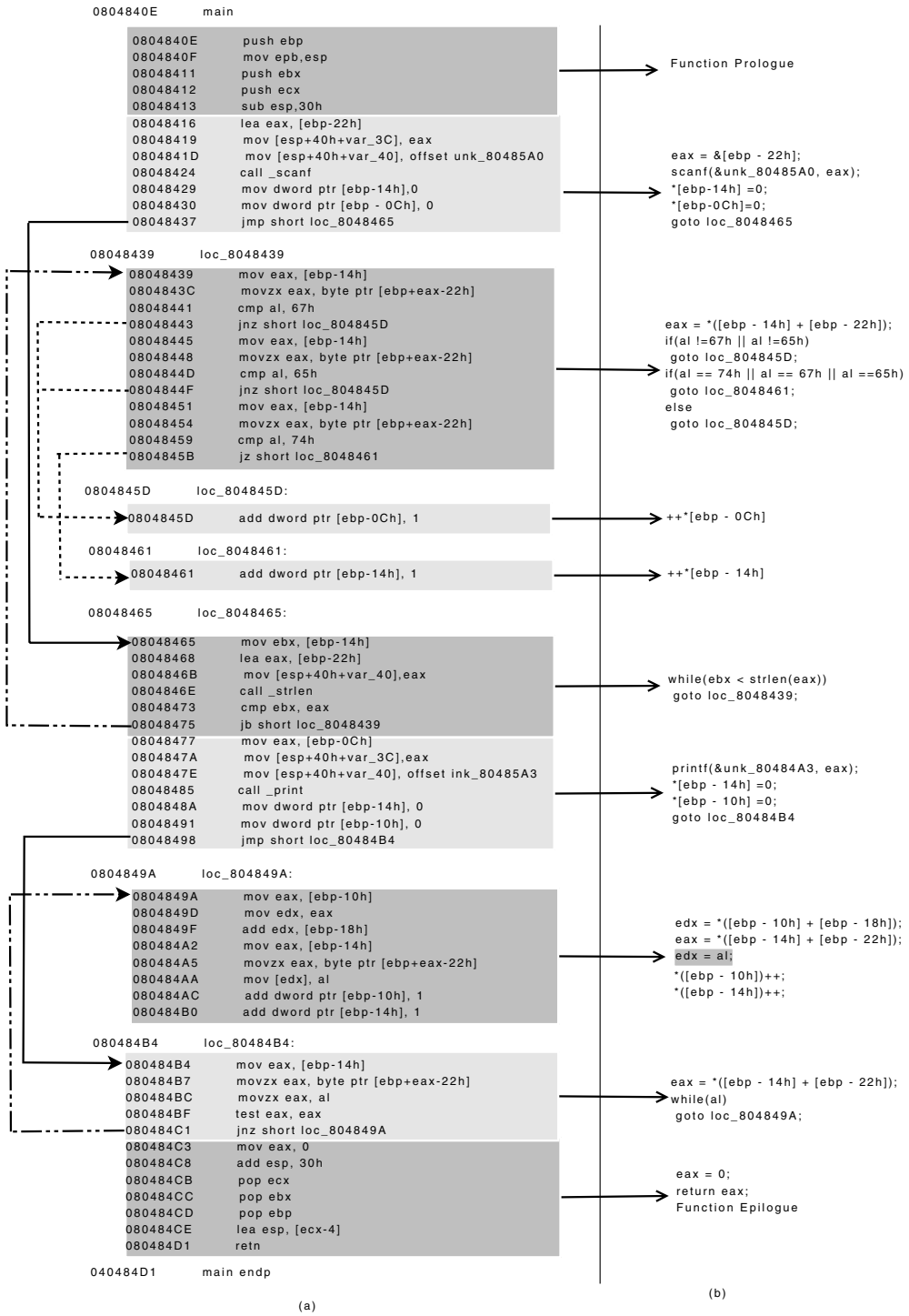

FIG. 3. Signature Generation Process for Assembly Instruction (a) Shows the assembly instruction of a vulnerable program where bold lines denote direct control transfer, doted lines denotes conditional statement (If-else) and, bold-dotted lines denotes looping statement (for,while), (b) Intermediate Representation of 'a' showing the vulnerability condition in the shaded region.

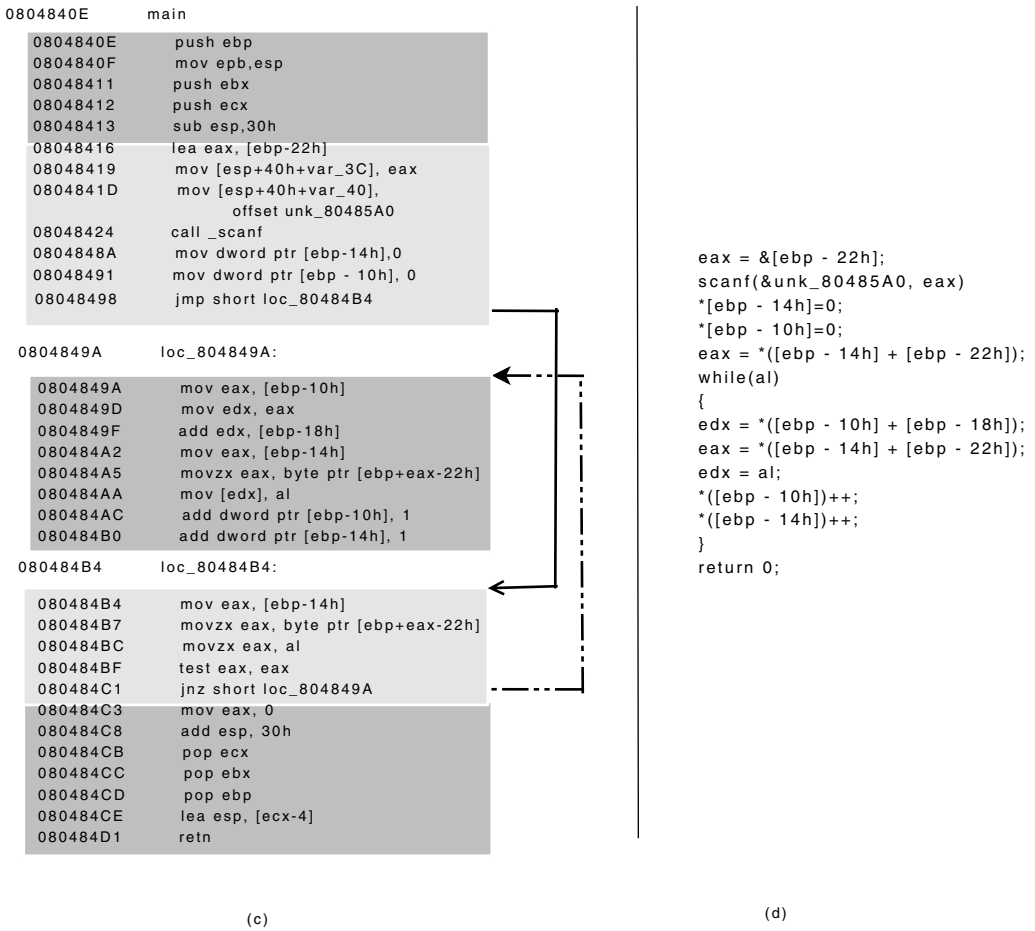

FIG. 4. Signature Generation Process for Assembly Instruction (c) vulnerability signature based on the vulnerability condition shown in 'b' (d) Pseudo Code of the vulnerability signature.

4. Implementation and Evaluation

In this section, we briefly describe our prototype implementation and describe the results of our experimental evaluation.

4.1. Implementation

We implemented and evaluated our prototype on Windows operating system.Our evaluation was performed on a Pentium 4 with a clock speed of 3.2GHz, equipped with a physical memory of 512MB and running Windows XP. Our implementation is divided into two components. The first component is a *translation engine* which interfaces directly with x86 binary and is responsible for parsing the binary (EXE and ELF), reading in the assemble, and converting it into *vulnerability signature*. The x86 executable is first disassembled using IDAPro[22]. To generate the IR, we have written plug-ins to IDAPro that perform additional analysis such as, identifying no-return functions and variable name extraction. In the second component, we translate IR into vulnerability signature and, use Understand C++(UDC) [49] to analyze the intermediate representation. UDC is a tool for code understanding and code inspection that supports the API for creating a vulnerability signature, as well as accessing other information stored in UDC's intermediate representations (IRs).

4.2. Evaluation

For evaluation purposes we have chosen only stripped binaries and generated signatures for them as these are the most common form of programs that may be available

to users. We have performed evaluation on eight vulnerable programs, four of which are linux-based vulnerable applications ATPhttpd(web server), ghttpd(web server), passlogd(syslog message sniffer), tcpflow and, four of which are Windows based DLLs, netapi32.dll, opera.dll, dxtmsft.dll and mshtml.dll. The summary of signature generation time and signature sizes in terms of lines of code (LOC) are shown in Table 3.

Linux Based Vulnerable Applications

- **ATPhttpd.** ATPhttpd is a web-server written in C [50] and version 0.4b is vulnerable to stack-smashing attack. This vulnerability can be exploited by requesting a file name that does not exist on the server. The vulnerability point is the call to *sprintf* from the function *http_send_error* using a large crafted input which can, say, overwrite a return address. Our technique generates the *vulnerability signature* in 1.399 seconds. The vulnerability signature for ATPhttpd is shown in Figure 5.
- **Ghttpd.** Ghttpd, a web server written in C, contains a stack-based overflow in version 1.4.3 [51]. Ghttpd calls a logging function *Log* on every request but fails to check the boundary of its local buffer when storing the log message *s* using *vsprintf*. The vulnerability can be exploited using entirely disjoint program execution paths. The vulnerability point is the call to *vsprintf* inside *Log* function. The vulnerability condition is satisfied by the parameters passed to *vsprintf*. The call tree of the vulnerability signature for Ghttpd is shown in Figure 6(b)
- **Passlogd.** Passlogd [52] is an all purpose sniffer used to capture syslog messages and is vulnerable to a stack-based buffer overflow. The vulnerability is present in *sl_parse* function. Passlogd fails to check the buffer boundary when searching for a delimiting '>', resulting in a stack overflow. The vulnerability point for this program is an assignment statement inside *sl_parse: level[j] = pkt[i]*. The vulnerability condition is the address calculated by the expression *level[j] = pkt[i]* which is the address of *sl_parse*'s frame pointer.
- **Tcpflow.** Tcpflow [53] is a network monitoring tool that records TCP sessions. Tcpflow version 2.0 contains a format string vulnerability. The vulnerability is present in *print_debug_message* function, where the error message are passed as the format string to the *vfprintf*() call. The vulnerability point is the call to the *vsprintf*() inside *print_debug_message* function. The vulnerability condition are satisfied by the parameters passed to *vsprintf*.

4.3. Vulnerable Windows DLLs

We conducted a survey of patches released from Microsoft in 2006 and found 84% of the security-related updates were changes in shared libraries. In this section, we have analyzed vulnerabilities in four Windows DLLs namely "netapi32.dll", "opera.dll", "dxtmsft.dll" and "mshtml.dll".

- **netapi32.dll.** `W32.Downadup` [54] exploits the vulnerability in Microsoft Windows Server Service's RPC handling that results in Remote Code Execution (described in the Microsoft Security Bulletin MS08-067). The Vulnerability is within the Server Service which can be exposed as either `Services.exe` or `Svchost.exe`. The process `svchost.exe` has a named pipe with the name `'srvsvc'` (Server Service handle). Also there is a dll called `srvsvc.dll`. Dissecting this dll shows that it exports a function called '_NetpwPathCanonicalize'. This function takes three wide character strings and three long values. To canonicalize the path feed it calls a function exported by `netapi32.dll` called `sub_5B86A51A` of the Windows XP SP3 platform.

  In the case of the recent Vulnerability in Windows XP SP3 Service `(MS08-067)` [55], the problem lies in the function sub_5B86A51A of netapi32.dll. This function strips out the preceding directory entries with each "/:/". It does this by finding where a "/:/" is and then stepping backwards up to the backslash that starts before it, and copying the string. It keeps the 0x5c or "/" in a stack and popping out one at a time as it jumps backward to the previous 0x5c. While doing this, when it encounters that there are no more prior directory entries, it jumps back to a memory location that is being used by the RPC Server Service which results in a situation that is definitely `EXPLOITABLE` and `WORMABLE` [56]. The call tree[1] generated for this vulnerability is shown in Figure 7.
- **dxtmsft.dll.** We next evaluated our technique on web browser vulnerabilities. The client-end web browser vulnerabilities come from the Month of Browser Bugs (MOBB) website [57]. These bugs are scripting vulnerabilities in Internet Explorer on Windows XP with Service Pack 2 installed, usually due to mishandled memory. Modern scripting languages extend basic HTML with the ability to call native methods on the browser's computer, e.g., ActiveX controls. When the web-browser renders a page containing a script, the script invokes

[1] A `Call Tree` is a functional flow graph which contains all the functions called in each function

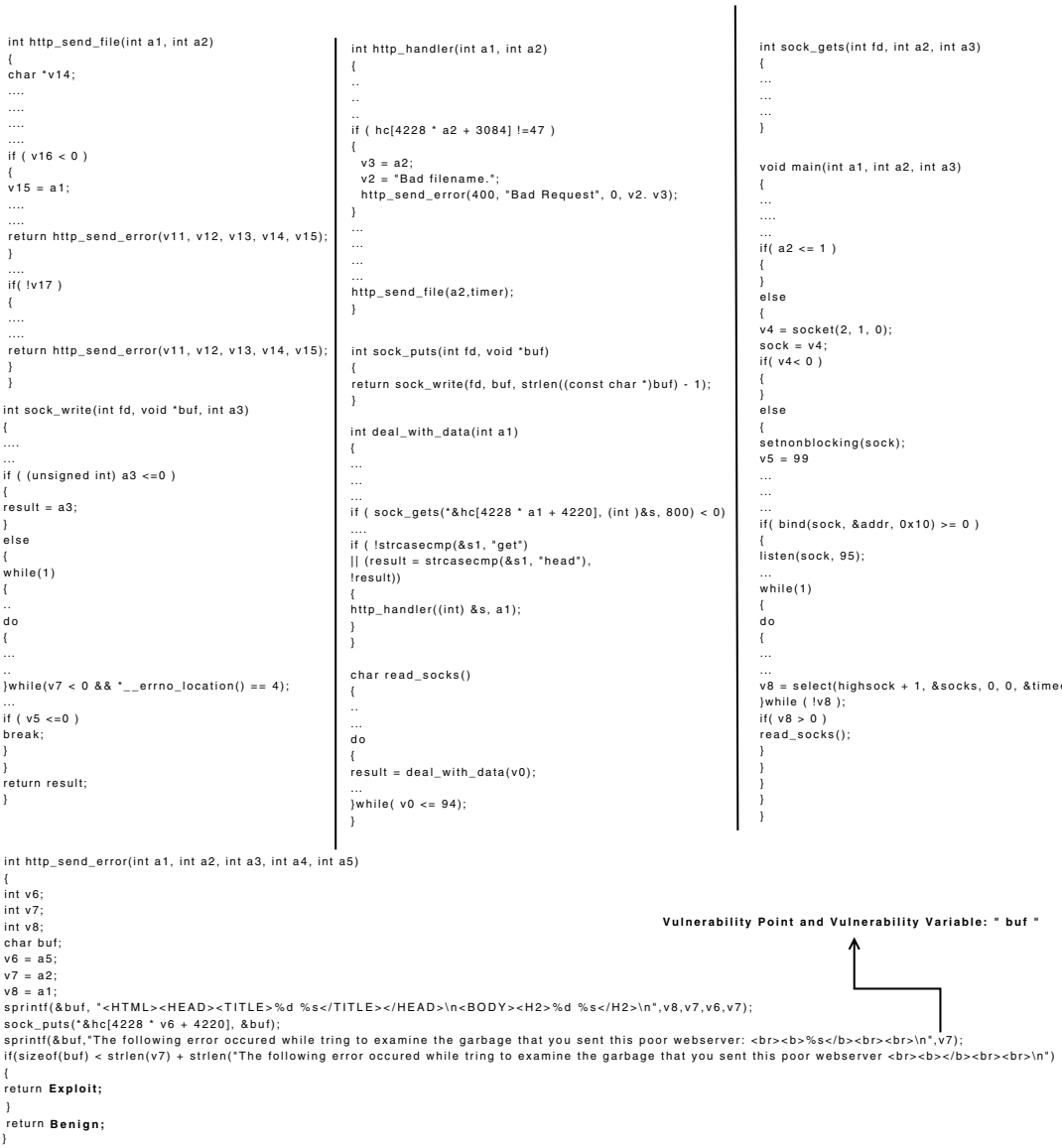

FIG. 5. Vulnerability Signature for ATPhttpd given the Vulnerability Condition at http_send_error function

the native method, which is executed with the privileges of the browser and can result in malicious behavior. We evaluated our technique on `MOBB 13` which contains denial-of-service vulnerability which arises when a script does not properly initialize all data-structures. The problem with this bug is that when we set the transition property of this object triggers a NULL dereference. MOBB 13 is based upon a module which allows PowerPoint-like transitions for images on a web page. A script calls the method `put_Transition(object, int)` where the second `int` argument specifies the transition to use. The object passed in is assumed to have a table of virtual functions, one for each possible transition which can be specified for the `int` argument. If the object does not have that transition defined, the corresponding virtual function is NULL. However, `put_transition` does not check for NULL, and a subsequent de-reference crashes the browser. The vulnerability point, as specified in MOBB 13, is in `dxtmsft.dll` at `CDXTRevealTrans::put`

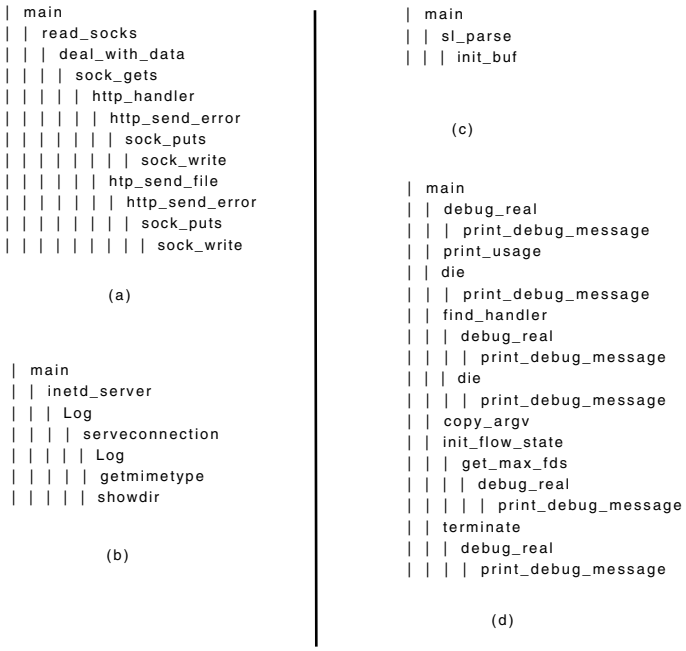

FIG. 6. Call Tree for the vulnerable program (a) ATPhttpd w.r.t vulnerability function *http_send_error*, (b) Ghttpd w.r.t *Log*, (c) Passlogd w.r.t *sl_parse* and, (d) Tcpflow w.r.t *print_debug_message*

`Transition+0x3a`. The vulnerability condition is EAX == 0 (since the EAX register holds the address at the vulnerability point) for MOBB 13.

- **mshtml.dll.** Next, we consider the `MS08-073` Microsoft Vulnerability which could allow remote code execution if a user views a specially crafted Web page using Internet Explorer 5.01. An attacker who successfully exploited this vulnerability could gain the same user rights as the logged-on user. On Internet Explorer 5.01 a function return address can be overwritten with attacker controlled data which results in an exploitable condition. MSHTML.DLL is the shared library for parsing HTML in Internet Explorer and related applications. The call tree generated for mshtml.dll is shown in Figure 4(b).
- **opera.dll.** We evaluated our technique on `Opera Browser` [20]. The vulnerability is present in Opera version 9 on a fully-patched Windows XP SP2 system. A memory corruption issue can be triggered by setting the background property of any DHTML element to a long HTTPS URL. The vulnerability is present in sub_67BEFB86 function and the vulnerability variable being v5.

4.4. Exploit Detection Speed

We measured the exploit detection performance of our signatures on both, malicious inputs and on benign inputs. We supplied the inputs to the signature S and verified whether the correct result (either EXPLOIT or BENIGN) was obtained. We report the average of 1000 runs on each input and show the results in Table 4. Here, we also compare our approach with Vulnerability State Machine (VSM) signatures from [17]. The "Matching attacks" shows the matching time on exploit inputs, which were generated from publicly available exploits and hand-crafted variants. The column "Matching benign" shows the matching time on benign inputs. It can be seen that our approach can detect exploits faster (by orders of magnitude) than the VSM based signatures. In Table 5 we compare signature generation time, size of signatures and compilation time across the other existing approach [17]. We conclude our approach generates not only accurate signatures but also faster signatures making them amenable to deployment in high-speed applications.

```
int * sub_5B863D20(int a1, int a2, int a3)
{
  a3 = a1;
  a2 = -1;
}

int sub_5B8679E0( int hMem)
{
  ..
  result = LocalFree(hMem);
  return result;
}

int sub_5B868FB7( int uBytes)
{
 ..
 ..
 ..
return result;
}

int I_NetPathCanonicalize(int a1,..., int a8)
{
 ..
 sub_5B863D20(a1,..);
 ..
 sub_5B8A4571(a1, a1 - 548,..., v14);
 ..
 NetpwPathCanonicalize((char *)v9, ..);
}

int NetpwPathCanonicalize(char *a1, ..., int a6)
{
 ..
 ..
 result  = sub_5B86A430(v6, a1, a2, v11, 0);
..
}


int sub_5B86A51A(const char *a1)
{
 ..
 ..
 sub_5B890F75(v8, ( int)(v16 - v8) >> 1, v6 + 4);
 ..
 ..
 sub_5B890F75(v5, (int)( v16 - v5) >> 1, v2);
}
```

```
int NetpwPathCompare(char *a1,...)
{
 ..
 if( .. )
 sub_5B8679E0(v5);
 ..
 else
  {
    v7 = sub_5B868FB7(0x208u);
    ..
    sub_5B679E0(v5);
    v9 = (char *)NetpwPathCanonicalize(a1, ...);
    ..
  }
}


int sub_5B88FD96(char a1, ...., int a8)
{
  ..
  ..
  v24 = NetpwPathCanonicalize(v22, ...);
 ..
}


int sub_5B890F75(int a1, int a2, int a3)
{
 if( .. )
  result = sub_5B890EB6(a1, a2, a3);
else
  result = -2147024809;
}


int sub_5B86A430(int a1, char *a2, ....)
{
 ..
 ..
 wcscpy(&v18, (const char *)v7);
 ..
 ..
 wcscat(&v18, v5);
}
```

```
int sub_5B8A4571(int a1, .. ,int a8)
{
  ..
  sub_5B863D20(a1, (int)&dword_5B8A4648, 24);
  ..
 v9 = sub_5B8A532B(....);
  ..
}

CLIENT_CALL_RETURN  sub_5B8A532D(int a1, ..)
{
  return NdrClientCall2(.....);
}

int I_NetListCanonicalize(int a1, int a2, ...,int a10)
{
 ..
 ..
 sub_5B863D20(a1, (int)&dword_5B87E0E8, 580);
 ..
 ..
sub_5B88FD96(**char **)(a1 - 556),*(char **)(a1 - 552),
                  v12, *(a1 + 24),*(a1 - 568), v11,
                  *(a1 + 36), *(a1 + 40));
..
}

int I_NetPathCompare(int a1, ...)
{
  ..
  sub_5B86D20(a1, (int)&dword_5B87E210, 552);
  ..
  if(..)
  {
    if( *(a1 - 560) == -1)
       sub_5B8A4659(a1, a1 - 548, v8, v7, v9, v10);
    else
       NetpwPathCompare((char *)v8, ......);
  }
}
```

FIG. 7. Complete Vulnerability Signature for netapi32.dll windows library

```
| I_NetPathCanonicalize
| | sub_5B863D20
| | sub_5B8A4571
| | | sub_5B863D20
| | | | sub_5B8A532D
| | NetpwPathCanonicalize
| | | sub_5B86A430
| sub_5B86A51A
| | sub_5B890F75
| I_NetListCanonicalize
| | sub_5B863D20
| | sub_5B88FD96
| | | NetpwPathDanonicalize
| | | | sub_5B86A430
| I_NetPathCompare
| | sub_5B863D20
| | NetpwPathCompare
| | | sub_5B8679E0
| | | sub_5B868FB7
| | | NetpwPathCanonicalize
| | | | sub_5B86A430
```

(a)

```
| sub_63713CB1
| | sub_6370CB24
| | | sub_6366CF22
| | | | sub_6366CDFA
| | | | sub_6385B08D
| | | | sub_6375B124
| | | | | sub_6385B3A3
| | | | | sub_6375AE25
| sub_63717718
| | sub_6370CB24
| | | sub_6366CF22
| | | | sub_6366CDFA
| | | | sub_6375B08D
| | | | sub_6375B124
| | | | | sub_6375B3A3
| | | | | sub_6375AE25
```

(b)

FIG. 8. Call Tree for Windows Library vulnerability (a) Netapi32.dll w.r.t *sub_5B86A51A* and, (b) Mshtml.dll w.r.t

Table 3. Signature Generation Performance (in seconds) and Size (in lines of code, LOC)

Vulnerable Program	Time to Compute IR	LOC in IR	Time to Compute Signature	LOC in Signature	Total Time
ATPhttpd	0.7656	1423	0.6343	315	1.3999
ghttpd	0.6249	1168	0.5250	353	1.1499
passlog(d)	0.651	7653	0.2434	49	0.8944
tcpflow	0.6716	1421	0.5875	274	1.2591
netapi32.dll	64.2340	58175	58.9062	408	123.1402
mshtml.dll	490.4058	505824	472.6590	370	963.0648
dxtmsft.dll	51.5465	56371	3.9531^2	27	55.4996

Table 4. Signature Matching Performance (seconds)

Vulnerable Program	Matching Benign		Matching Attacks	
	VSM [17]	**Our Approach**	**VSM** [17]	**Our Approach**
ATPhttpd	0.00015	0.00010	0.00113	0.00020
Ghttpd	0.000165	0.00013	0.00025	0.00023
Passlogd	0.00003	0.000023	0.00051	0.00003

Table 5. Time (in seconds) comparison with VSM based Signatures

Vulnerability Program	Signature Size		Creating Signature		Compile Signature	
	VSM	**Our Approach**	**VSM** [17]	**Our Approach**	**VSM** [17]	**Our Approach**
ATPhttpd	9087	315	0.676	0.6343	0.88	0.124
Ghttpd	12697	353	0.549	0.5250	0.61	0.091
Passlogd	2101	49	0.273	0.2434	0.050	0.060

5. Conclusion and Future Work

In this paper, we presented a novel backtracing approach for automatically creating the *vulnerability signature* by using the vulnerability condition. Our approach allows us to generate compact symbolic signatures which are sound and accurate. We have evaluated our signatures on a number of known programs and exploits. Our results show a faster detection rate, of the order of 1.5 to 7 times faster than the best known approach so far. The main feature of our approach is that it works for programs with source code available, on binary executables and on shared libraries. One of the key challenges in this area is to be able generate patches that do not require extensive regression testing. In its present form, extending the current algorithm to generate patches for production software is a non-trivial task. We are currently working on understanding the mechanics behind the automation of patch generation that avoids tedious regression testing cycles. We surmise that the signature generation approach gives us valuable insights into understanding the program behavior which can help towards solving this problem.

References

[1] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. Autopag: towards automated software patch generation with source code root cause identification and repair. In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 329–340, New York, NY, USA, 2007. ACM.

[2] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony I. T. Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: end-to-end containment of internet worms. In *SOSP*, pages 133–147, 2005.

[3] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium*, pages 271–286, 2004.

[4] Zhichun Li, Manan Sanghi, Yan Chen, Ming-Yang Kao, and Brian Chavez. Hamsa: Fast signature generation for zero-day polymorphicworms with provable attack resilience. In *IEEE Symposium on Security and Privacy*, pages 32–47, 2006.

[5] Zhenkai Liang and R. Sekar. Fast and automated generation of attack signatures: a basis for building self-protecting servers. In *ACM Conference on Computer and Communications Security*, pages 213–222, 2005.

[6] Christian Kreibich and Jon Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *Computer Communication Review*, 34(1):51–56, 2004.

[7] James Newsome, Brad Karp, and Dawn Xiaodong Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, pages 226–241, 2005.

[8] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *OSDI*, pages 45–60, 2004.

[9] Joseph Tucek, James Newsome, Shan Lu, Chengdu Huang, Spiros Xanthos, David Brumley, Yuanyuan Zhou, and Dawn Xiaodong Song. Sweeper: a lightweight end-to-end system for defending against fast worms. In *EuroSys*, pages 115–128, 2007.

[10] Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, and Christopher Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *ACM Conference on Computer and Communications Security*, pages 223–234, 2005.

[11] David Brumley, Li-Hao Liu, Pongsin Poosankam, and Dawn Xiaodong Song. Design space and analysis of worm defense strategies. In *ASIACCS*, pages 125–137, 2006.

[12] David Moore, Colleen Shannon, Geoffrey M. Voelker, and Stefan Savage. Internet quarantine: Requirements for containing self-propagating code. In *INFOCOM*, 2003.

[13] James Newsome, Brad Karp, and Dawn Xiaodong Song. Paragraph: Thwarting signature learning by training maliciously. In *RAID*, pages 81–105, 2006.

[14] Roberto Perdisci, David Dagon, Wenke Lee, Prahlad Fogla, and Monirul I. Sharif. Misleadingworm signature generators using deliberate noise injection. In *IEEE Symposium on Security and Privacy*, pages 17–31, 2006.

[15] David Brumley, James Newsome, Dawn Xiaodong Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*, pages 2–16, 2006.

[16] David Brumley, Hao Wang, Somesh Jha, and Dawn Song. Creating vulnerability signatures using weakest preconditions. In *CSF*, pages 311–325, 2007.

[17] David Brumley, Zhenkai Liang, James Newsome, and Dawn Song. Towards practical automatic generation of multipath vulnerability signatures. technical report cmu-cs-07-150, carnegie mellon university school of computer science. 2007.

[18] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 85–96, New York, NY, USA, 2004. ACM.

[19] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, 2005.

[20] Opera browser. http://www.opera.com/.

[21] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[22] Idapro disassembler. http://www.hex-rays.com/idapro/.

[23] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[24] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 9–9, Berkeley, CA, USA, 2006. USENIX Association.

[25] Jedidiah R. Crandall, Zhendong Su, Shyhtsun Felix Wu, and Frederic T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *ACM Conference on Computer and Communications Security*, pages 235–248, 2005.

[26] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *SIGCOMM*, pages 193–204, 2004.

[27] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP*, pages 91–104, 2005.

[28] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, Berkeley, CA, USA, 1998. USENIX Association.

[29] Stack shield - a "stack smashing" technique protection tool for linux. 2006. http://www.angelfire.com/sk/stackshield/.

[30] H. Etoh. Gcc extension for protecting applications from stack-smashing attacks. 2006. http://www.trl.ibm.com/projects/security/ssp/.

[31] T. Tsai and N. Singh. Libsafe: Transparent System-Wide Protection against Buffer Overflow Attacks. 2002.

[32] Kumar Avijit, Prateek Gupta, and Deepak Gupta. TIED, Libsafeplus: Tools for Runtime Buffer Overflow Protection. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2004. USENIX Association.

[33] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. Formatguard: Automatic Protection from Printf Format String Vulnerabilities. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2001. USENIX Association.

[34] Pankaj Kohli and Bezawada Bruhadeshwar. Formatshield: A Binary Rewriting Defense against Format String Attacks. In *ACISP*, pages 376–390, 2008.

[35] T. Robbins. Libformat. http://www.wiretapped.net/ fyre/ software/libformat.html.

[36] Jin Ho You, Seong Chae Seo, Young Dae Kim, Jun Yong Choi, Sang Jun Lee, and Byung Ki Kim. Kimchi: A Binary Rewriting Defense Against Format String Attacks. In *WISA*, pages 179–193, 2005.

[37] Wei Li and Tzi cker Chiueh. Automated Format String Attack Prevention for Win32/X86 Binaries. In *ACSAC '07: Proceedings of the 23nd Annual Computer Security Applications Conference*, pages 398–409, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[38] Michael F. Ringenburg and Dan Grossman. Preventing Format-String Attacks via Automatic and Efficient Dynamic Checking. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 354–363, New York, NY, USA, 2005. ACM.

[39] PaX Team. PaX Address Space Layout Randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[40] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2003. USENIX Association.

[41] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2005. USENIX Association.

[42] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2001. USENIX Association.

[43] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A Theory of Type Qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 192–203, New York, NY, USA, 1999. ACM.

[44] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, Berkeley, CA, USA, 2002. USENIX Association.

[45] Pete Broadwell, Matt Harren, and Naveen Sastry. Scrash: A System for Generating Secure Crash Information. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2003. USENIX Association.

[46] James Newsome and Dawn Xiaodong Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *NDSS*, 2005.

[47] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *ISCC '06: Proceedings of the 11th IEEE Symposium on Computers and Communications*, pages 749–754, Washington, DC, USA, 2006. IEEE Computer Society.

[48] Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society.

[49] S. Toolworks. Maintenance, understanding, metrics and documentaion tools for ada,c,c++,java,and fortan. 2006. http://www.scitools.com/.

[50] Yann Ramin. Atphttpd. http://www.redshift.com/yramin/atp/atphttpd/.

[51] pyramid rphushmail.com. Ghttpd log() function buffer overflow vulnerability. http://www.securityfocus.com/ Bugtraq ID 5960.

[52] dong h0un. Passlogd sl_parse remote buffer overflow vulnerability. http:// www.securityfocus.com/ Bugtraq ID 7261.

[53] Jeremy Elson. Tcpflow format string vulnerability. http:// www.securityfocus.com/Bugtraq ID 8366.

[54] W32.downadup. http://www.symantec.com/security_response/writeup.jsp?docid=2008-112203-2408-99&tabid=2.

[55] Microsoft security bulletin ms08-067. http://www.microsoft.com/technet/security/Bulletin/MS08-067.mspx.

[56] Decompiling the vulnerable function for ms08-067. http:// www.phreedom. org/blog/2008/decompiling-ms08-067/.

[57] Mobb vulnerabilities analysis. http://browserfun.blogspot.com//