A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows*

Ramkumar Chinchani[1] and Eric van den Berg[2]

[1] University at Buffalo (SUNY), Buffalo, NY 14260, USA
`rc27@cse.buffalo.edu`
[2] Applied Research, Telcordia Technologies, Piscataway, NJ 08854
`evdb@research.telcordia.com`

Abstract. A common way by which attackers gain control of hosts is through remote exploits. A new dimension to the problem is added by worms which use exploit code to self-propagate, and are becoming a commonplace occurrence. Defense mechanisms exist but popular ones are signature-based techniques which use known byte patterns, and they can be thwarted using polymorphism, metamorphism and other obfuscations. In this paper, we argue that exploit code is characterized by more than just a byte pattern because, in addition, there is a definite control and data flow. We propose a fast static analysis based approach which is essentially a litmus test and operates by making a distinction between data, programs and program-like exploit code. We have implemented a prototype called *styx* and evaluated it against real data collected at our organizational network. Results show that it is able to detect a variety of exploit code and can also generate very specific signatures. Moreover, it shows initial promise against polymorphism and metamorphism.

1 Introduction and Motivation

External attackers target computer systems by exploiting unpatched vulnerabilities in network services. This problem is well-known and several approaches have been proposed to counter it. Origins of a vulnerability can be traced back to bugs in software, which programming language security approaches attempt to detect automatically. [37,10]. However, due to technical difficulties involved in static analysis of programs [25,32], not all bugs can be found and eliminated. An alternative approach is to detect attacks at runtime either via code instrumentations [18,13] or intrusion detection [15]. But runtime checks may cause significant overheads as an undesirable side-effect.

An orthogonal approach which complements these techniques in preventing remote attacks involves detecting exploit code inside network flows. An important advantage of this approach is that it is proactive and countermeasures can be taken even before the exploit code begins affecting the target program.
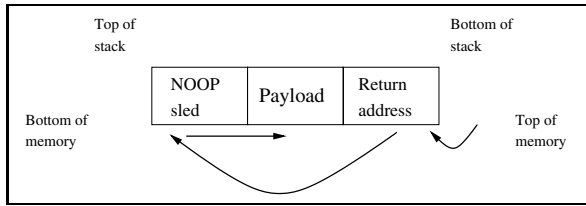
Fig. 1. General structure of exploit code

Figure 1 shows the structure of a typical exploit code, which consists of three distinct components - 1) a *return address block*, 2) a *NOOP sled*, and 3) the *payload*. The main purpose of such a construction is that when a function returns following a buffer overflow, the return address block directs execution on to the NOOP sled, which eventually reaches the payload.

The basic idea of exploit code detection inside network flows with the goal of preventing remote exploits is not new. Support for packet-level pattern matching has long been offered by network-based intrusion detection systems such as Snort and Bro, and detecting exploit code entails specifying the corresponding signature. While such systems are relatively easy to implement and perform well, their security guarantees are only as good as the signature repository. Evasion is simply a matter of operating outside a signature repository and this is achieved either by altering instructions or instruction sequence (metamorphism), encryption/decryption (polymorphism), or discovering an entirely new vulnerability and writing the corresponding exploit (zero-day exploit). As a rule of thumb, signatures must be *long* so that they are specific enough to reduce false positives which may occur when normal data accidentally matches exploit code signatures. Also, the number of signatures has to be *few* to achieve scalability because the signature matching process can become computationally and storage intensive. These two goals are seriously hindered by polymorphism and metamorphism and pose significant challenges for signature-based detection systems especially when automated toolkits are available [6,3].

Polymorphism and metamorphism affect the three components of exploit code differently. The payload component can be concealed to evade signature-based detection using either polymorphism and metamorphism, and therefore, is seldom the focus of detection. In an exploit code, the return address block and the NOOP sled are used to improve chances of success by accounting for the uncertainty regarding the vulnerable buffer such as its actual address in memory. Therefore, it is only reasonable to assume that polymorphic encryption cannot be applied to them and they must be in plain view. On the downside, the NOOP sled is still susceptible to metamorphism and the return address block may be too short to be useful. Consequently, although recently proposed techniques [16,35,31] for detection of exploit code have attempted to cope with polymorphism and metamorphism, there are shortcomings and some challenges remain. To summarize, signature-based detection techniques cannot provide all the answers and we must look elsewhere for more effective techniques.

Table 1. Some popularly targeted network services as reported by SANS [5], their port numbers and the general nature of network flows on the corresponding ports as observed empirically

Microsoft Windows		
Vulnerable service/program	**Port**	**Content Type**
IIS Webserver	80	Mostly data
Workstation Service	139, 445	Data
Remote Access Services	111, 137, 138, 139	Data
Microsoft SQL Server	1434	Data
Instant messaging (MSN, Yahoo, AOL)	1863, 5050, 5190-5193	Mostly data
GNU/Linux		
Vulnerable service/program	**Port**	**Content Type**
BIND	53	Data
Apache Webserver	80	Mostly data
pserver/Version Control	2401	Data
Mail Transport	25	Mostly data
SNMP	161	Data
Database Systems (Oracle, MySQL, PostgreSQL)	1521, 3306, 5432	Data

In this paper, we propose an approach which takes the viewpoint that the nature of communication to and from network services is predominantly or exclusively data and not executable code (see Table 1). Since remote exploits are typically executable code transmitted over a network, it is possible to detect exploits if a distinction can be made between data and executable code in the context of a network flow. One such exploit code indicator was proposed by Toth and Kruegel [35] wherein binary disassembly is performed over a network flow and a long sequence of valid instructions shows the presence of a NOOP sled. However, this scheme falls short, firstly because it is easily defeated by a metamorphic NOOP sled [16], and secondly, because it doesn't take into account information given away by branch instructions. Hence, mere binary disassembly is not adequate.

Exploit code, although not a full program, is very "program-like" and has a certain structure. Moreover, the code must achieve whatever goal was intended by the exploit code author through some sequence of executable instructions. Therefore, there is a definite data and control flow, and at least some of which must be in plain view. Our approach to exploit detection is to look for evidence of meaningful data and control flow, essentially focusing on both NOOP sled and payload components whenever possible. An important consequence of using a static analysis based approach is that it can not only detect previously unseen exploit code but is also more resilient to changes in implementation which exploit code authors employ to defeat signature-based techniques.

There are significant differences both in terms of goals and challenges faced between static analysis of programs and our approach. When performing static analysis, the goal is to reason about a program and answer the question: can program execution lead to unpredictable or malicious behavior? We face a different

problem, which is phrased as follows. Consider one or more executable code fragments with no additional information in terms of program headers, symbol tables or debugging information. At this point, we neither have a well-defined program nor can we trivially determine the execution entry point. Next, consider a set of network flows and arbitrarily choose both a flow as well as the location inside the flow where the code fragments will be embedded. Now, we ask the question: given a flow, can we detect whether a flow contains the program-like code or not? Also, if it does, can we recover at least majority of the code fragments, both for further analysis as well as signature generation? In other words, one challenge is to perform static analysis while recovering the code fragments without the knowledge of their exact location. The other is that the process must be efficient in order scale to network traffic. We show that this is possible albeit in a probabilistic sense.

The relevance of our work goes beyond singular exploits. Lately, there has been a proliferation of Internet worms and there is a strong relationship between the worm spread mechanism and remote exploits.

1.1 Connections Between Exploit Code and Worm Spread Mechanism

Following earlier efforts [33,43,29] in understanding worms which are self-propagating malware, several techniques have been proposed to detect and contain them. For a comprehensive overview of various types of worms, we recommend the excellent taxonomy by Weaver et al. [39].

As is the case with most security areas, there is an arms race unfolding between worm authors and worm detection techniques. For example, portscan detection algorithms [19,40] proposed to rapidly detect scanning worms can be eluded if hitlists are used and one such worm named Santy surfaced recently which used Google searches to find its victims. Table 2 is a compilation of a few representative worm detection algorithms, their working principles and worm counterexamples which can evade detection. In their taxonomy, Weaver et al. [39] had foreseen such possibilities and only within a year of this work, we are beginning to see the corresponding worm implementations. Moreover, with the availability of now mature virus and exploit authoring toolkits which can create stealthy code [6,3], a worm author's task is becoming increasingly easy.

The main point we want to make is that while the working principles specified in the second column of Table 2 are *sufficient* conditions for the presence of worm activity, they are not *necessary* conditions and the counterexamples in the third column support the latter claim. The necessary condition for a worm is *self-propagation* and so far this property has been realized primarily through the use of exploit code; a situation which is unlikely to change. Therefore, if an effective technique can be devised to detect exploit code, then we automatically get worm detection for free regardless of the type of the worm.

1.2 Contributions

There are two main contributions in this paper. As the first contribution, we propose a static analysis approach which can be used over network flows with

Table 2. A compilation of worm detection techniques, their working principles and counterexamples

Worm Detection Approach	Working Principle	Counterexample
Portscan Detection [19,41]	Scanning worms discover victims by trial-and-error, resulting in several failed connections	Histlist worms, e.g. Santy worm [1] used Google searches.
Distributed Worm Signature Detection [21]	Worm code propagation appears as replicated byte sequences in network streams	Polymorphic and metamorphic worms, e.g. Phatbot worm [27].
Worm/virus Throttle [36]	Rate-limiting outgoing connections slows down worm spread	Slow-spreading worms.
Network Activity-Based Detection [42]	Detect "S"-shaped network activity pattern characteristic of worm propagation	Slow-spreading worms.
Honeypots/Honeyfarms	Collections of honeypots fed by network telescopes, worm signatures obtained from outgoing/incoming traffic.	Anti-honeypot technology [23]
Statistics-Based Payload Detection [38]	Normal traffic has different byte-level statistics than worm infested traffic	Blend into normal traffic [22]

the aim of distinguishing data and program-like code. In this regard, we answer the following two questions.

How can the instruction stream of an exploit code be recovered without the knowledge of its exact location inside a network flow? The exact location of the exploit code inside a network flow depends on several factors, one of them being the targeted vulnerability, and since we have no prior information about the existence of vulnerabilities or lack thereof, we cannot make any assumptions. Nevertheless, Linn et al. [26] observed that Intel binary disassembly has a self-correcting property, that is, performing disassembly over a byte stream containing executable code fragments but without the knowledge of their location still leads to the recovery of a large portion of the executable fragments. Our approach also leverages this property and we present a more in-depth analysis to show that it is relevant even for network flows. Consequently, we have an efficient technique to recover the instruction stream, which although lossy, is sufficiently accurate to perform static analysis.

How can static analysis be performed with only a reasonable cost? Static analysis typically incurs a very high cost and is only suitable for offline analysis. On the other hand, our aim in using static analysis is only to the extent of realizing an exploit code indicator which establishes a distinction between data and

executable code. We analyze the instruction stream produced via binary disassembly using basic data and control flow, and look for a meaningful structure in terms a sequence of valid instructions and branch targets. Such a structure has a very low probability of occurrence in a random data stream. Since we use an abbreviated form of static analysis, the costs are reasonable, which makes it suitable for use in online detection. In the context of detection, false positives can occur when random data is mistaken for executable code, but this is highly unlikely. Also, an exploit code author may deliberately disguise executable code as data, leading to false negatives. This is a harder problem to solve and we pay attention to this aspect during algorithm design wherever relevant.

These two aspects in cohesion form the core of our exploit code detection methodology, which we call *convergent static analysis*. We have evaluated our approach using the Metasploit framework [3], which currently supports several exploits with features such as payload encryption and metamorphic NOOP sleds. We are interested mainly in evaluating effectiveness in detecting exploit code and resistance to evasion. Also, given the popularity of the 32-bit x86 processor family, we consider the more relevant and pressing problem of detecting exploit code targeted against this architecture.

As our second contribution, we describe the design and architecture of an network flow based exploit code detection sensor hinging on this methodology. Sensor deployment in a real-world setting raises several practical issues such as performance overheads, sensor placement and management. In order to gain insight into these issues, we have performed our evaluation based on traces (several gigabytes in size) collected from an 100Mbps enterprise network over a period of 1-2 weeks. The dataset consists of flows that are heterogeneous in terms of operating systems involved and services running on the hosts.

1.3   Summary of Results

As a primary exploit detection mechanism, our approach offers the following benefits over signature-based detection systems.

- It can detect zero-day and metamorphic exploit code. Moreover, it can also detect polymorphic code, but the mileage may vary.
- It does not incur high maintenance costs unlike signature-based detection systems where signature generation and updates are a constant concern.

While our approach can operate in a stand-alone manner, it can also complement signature-based detection systems, offering the following benefit.

- If signature-based detection is to be effective, then the signature repository has to be kept up-to-date; a practically impossible task without automated tools. Our approach, by virtue of its ability to separate data and exploit code, identify portions of a network flow which correspond to an exploit. Therefore, it also serves as a technique which can automatically generate precise and high quality signatures. This is particularly invaluable since significant effort goes into maintaining the signature repository.

The rest of the paper is organized as follows. Related work is discussed in Section 2. Our first contribution is presented in Section 3. The core exploit code detection mechanism is described in Section 4.

2   Related Work

The two broad areas which are relevant to our work are exploit code detection inside network flows and static analysis, and significant advances have been made in both these areas. We review and compare some of them to put our work in perspective.

Several research efforts have acknowledged these evasion tactics and proposed possible solutions to deal with them, but they have their limitations. Hittel [16] showed how a metamorphic sled can be constructed and in the same paper, developed Snort rules for detection; however, their number can be very large. Toth and Kruegel [35], also concentrating on the NOOP sled, went one step further. They used binary disassembly to find sequences of executable instructions bounded by branch or invalid instructions; hence, longer the sequence, greater the evidence of a NOOP sled. However, this scheme can be easily defeated by interspersing branch instructions among normal code [16], resulting in very short sequences. In our approach, although we perform binary disassembly, its purpose is to assist static analysis. Recently, Pasupulati et al. [31] proposed a technique to detect the return address component by matching against candidate buffer addresses. While this technique is very novel and perhaps the first to address metamorphic and polymorphic code, there are caveats. First, the return address component could be very small so that when translated to a signature, it is not specific enough. Secondly, even small changes in software are likely to alter buffer addresses in memory. Consequently, this approach runs into similar administrative overheads as existing signature-based detection systems. We do not focus on the return address component and changes in software do not impact our approach. Wang et al. [38] proposed a payload based anomaly detection system called PAYL which works by first training with normal network flow traffic and subsequently using several byte-level statistical measures to detect exploit code. But it is possible to evade detection by implementing the exploit code in such a way that it statistically mimics normal traffic [22].

Instruction recovery is central to static analysis and there are two general approaches - 1) *linear sweep*, which begins decoding from the first byte, and 2) *recursive traversal* [9], which follows instruction flow as it decodes. The first approach is straightforward with the underlying assumption that the entire byte stream consists exclusively of instructions. In contrast, the common case for our approach is the byte stream exclusively contains data. The second approach tries to account for data embedded among instructions. This may seem similar to our approach but the major difference is that the execution entry point must be known for recursive traversal to follow control flow. When the branch targets are not obvious due to obfuscations, then it is not trivial to determine control flow. To address this issue, an extension called *speculative disassembly* was proposed

by Cifuentes et al. [12], which as the name suggests attempts to determine via a linear sweep style disassembly whether a portion of the byte stream could be a potential control flow target. This is similar to our approach since the main idea is to reason whether a stream of bytes can be executable code. In general, all these approaches aim for accuracy but for our approach although accuracy is important, it is closely accompanied by the additional design goal of efficiency.

The differences between static analysis of malicious programs and exploit code inside network flows notwithstanding, there are lessons to be learnt from studies of obfuscation techniques which hinder static analysis as well as techniques proposed to counter them. Christodorescu et al. reported that even basic obfuscation techniques [11] can cause anti-virus scanners to miss malicious code. They go on to describe a technique to counter these code transformation using general representations of commonly occurring virus patterns. Linn et al. describe several obfuscation techniques [26] which are very relevant to our approach, such as embedding data inside executable code to confuse automatic disassembly. Kruegel et al. devised heuristics [24] to address some of these obfuscations. These algorithms tackle a much harder problem and aim for accuracy in static analysis, while our approach does not for reasons of efficiency and only partial knowledge being available.

3   Convergent Binary Disassembly

Static analysis of binary programs typically begins with disassembly followed by data and control flow analysis. In general, the effectiveness of static analysis greatly depends on how accurately the execution stream is reconstructed. This is still true in our case even if we use static analysis to distinguish data and executable code in a network flow rather than in the context of programs. However, this turns out to be a significant challenge as we do not know if a network flow contains executable code fragments and even if it does, we do not know where. This is a significant problem and it is addressed in our approach by leveraging the self-correcting property of Intel binary disassembly [26]. In this section, we perform an analysis of this property in the context of network flows.

3.1   Convergence in Network Flows

The self-correcting property of Intel binary disassembly is interesting because it tends to converge to the same instruction stream with the loss of only a few instructions. This appears to occur in spite of the network stream consisting primarily of random data and also when disassembly is performed beginning at different offsets. These observations are based on experiments conducted over network flows in our dataset. We considered four representative types of network flows - HTTP (plain text), SSH (encrypted), X11 (binary) and CIFS (binary). As for the exploit code, we used the Metasploit framework to automatically generate a few instances. We studied the effects of binary disassembly by varying the offsets of the embedded exploit code as well as the content of the network flow.
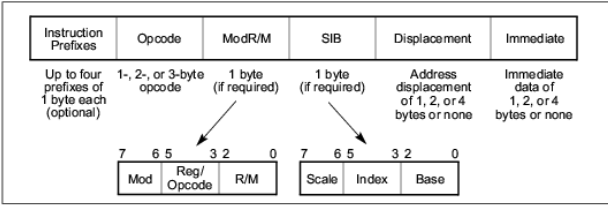
Fig. 2. General IA-32 instruction format

Convergence occurred in *every* instance but with different number of incorrectly instructions, ranging from 0 to 4 instructions.

The phenomenon of convergence can be explained by the nature of the Intel instruction set. Since Intel uses a complex instruction set computer architecture, the instruction set is very dense. Out of the 256 possible values for a given start byte to disassemble from, only one (`0xF1`) is illegal [2]. Another related aspect for rapid convergence is that Intel uses a variable-length instruction set. Figure 2 gives a overview of the general instruction formation for the IA-32 architecture [2]. The length of the actual decoded instruction depends not only on the opcode, which may be 1-3 bytes long, but also on the directives provided by the prefix, ModR/M and SIB bytes wherever applicable. Also note that not all start bytes will lead to a successful disassembly and in such an event, they are decoded as a data byte.

3.2   Analysis

We give a more formal analysis for this phenomenon. Given a byte stream, let's assume that the actual exploit code is embedded at some offset $x = 0, 1, 2, \ldots$. Ideally, binary disassembly to recover the instruction stream should begin or at least coincide at x. However, since we do not know x, we start from the first byte in the byte stream. We are interesting in knowing how soon after x does our disassembly synchronize with the actual instruction stream of the exploit code.

To answer this question, we model the process of disassembly as a random walk over the byte stream where each byte corresponds to a state in the state space. Disassembly is a strictly forward-moving random walk and the size of each step is given by the length of the instruction decoded at a given byte. There are two random walks, one corresponding our disassembly and the other corresponding to the actual instruction stream. Note that both random walks do not have to move simultaneously nor do they take the same number of steps to reach the point where they coincide.

Translating to mathematical terms, let $L = \{1, \ldots, N\}$ be the set of possible step sizes or instruction lengths, occurring with probabilities $\{p_1, \ldots, p_N\}$. For the first walk, let the step sizes be $\{X_1, \ldots, |X_i \in L\}$, and define $Z_k = \sum_{j=1}^{k} X_j$. Similarly for the second walk, let step sizes be $\{\tilde{X}_1, \ldots, |\tilde{X}_i \in L\}$ and $\tilde{Z}_k = \sum_{j=1}^{k} \tilde{X}_j$. We are interested in finding the probability that the random walks $\{Z_k\}$ and $\{\tilde{Z}_k\}$ intersect, and if so, at which byte position.

One way to do this, is by studying 'gaps', defined as follows: let $G_0 = 0$, $G_1 = |\tilde{Z}_1 - Z_1|$. $G_1 = 0$ if $\tilde{Z}_1 = Z_1$, in which case the walks intersect after 1 step. In case $G_1 > 0$, suppose without loss of generality that $\tilde{Z}_1 > Z_1$. In terms of our application: $\{Z_k\}$ is the walk corresponding to our disassembly, and $\{\tilde{Z}_k\}$ is the actual instruction stream. Define $k_2 = \inf\{k : Z_k \geq \tilde{Z}_1\}$ and $G_2 = Z_{k_2} - \tilde{Z}_1$. In general, Z and $\tilde{Z}$ change roles of 'leader' and 'laggard' in the definition of each 'gap' variable G_n. The $\{G_n\}$ form a Markov chain. If the Markov chain is irreducible, the random walks will intersect with positive probability, in fact at the first time the gap size is 0. Let $T = \inf\{n > 0 : G_n = 0\}$ be the first time the walks intersect. The byte position in the program block where this intersection occurs is given by $Z_T = Z_1 + \sum_{i=1}^{T} G_i$.

In general, we do not know Z_1, our initial position in the program block, because we do not know the program entry point. Therefore, we are most interested in the quantity $\sum_{i=1}^{T} G_i$, representing the number of byte positions after the dissassembly starting point that synchronization occurs.

Using partitions and multinomial distributions, we can compute the matrix of transition probabilities $p_n(i, j) = P(G_{n+1} = j | G_n = i)$ for each $i, j \in \{0, 1, \ldots N - 1\}$. In fact $p_n(i, j) = p(i, j)$ does not depend on n, i.e. the Markov chain is homogeneous. This matrix allows us e.g. to compute the probability that the two random walks will intersect n positions after disassembly starts.

The instruction length probabilities $\{p_1, \ldots, p_N\}$ required for the above computations are dependent on the byte content of network flows. The instruction length probabilities were obtained by disassembly and statistical computations over the same network flows chosen during empirical analysis (HTTP, SSH, X11, CIFS). In Figure 3, we have plotted the probability $P(\sum_{i=1}^{T} G_i > n)$, that intersection (synchronization) occurs beyond n bytes after start of disassembly, for $n = 0, \ldots, 99$.
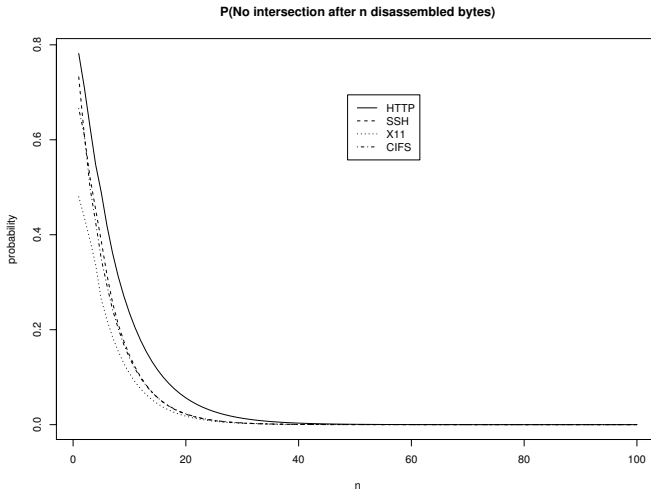
Fig. 3. Probability that the walk corresponding to our disassembly and the actual instruction flow will not have intersected after n bytes

It is clear that this probability drops fast, in fact with probability 0.95 the disassembly walk' and the 'program walk' will have intersected on or before the 21st (HTTP), 16th (SSH), 15th (X11) and 16th (CIFS) byte respectively, after the disassembly started. On average, the walks will intersect after just 6.3 (HTTP), 4.5 (SSH), 3.2 (X11) and 4.3 (CIFS) bytes respectively.

4   Static Analysis Based Detection

From a security standpoint, static analysis is often used to find vulnerabilities and related software bugs in program code. It is also used to determine if a given program contains malicious code or not. However, due to code obfuscation techniques and undecidability of aliasing, accurate static analysis within reasonable time bounds is a very hard problem. On one hand, superficial static analysis is efficient but may lead to poor coverage, while on the other, a high accuracy typically entails a prohibitively large running time.

4.1   Working Premise

In our approach, we use static analysis over network flows, and in order to realize an online network-based implementation, efficiency is an important design goal. Normally, this could translate to poor accuracy, but in our case we use static analysis only to devise a process of elimination. which is based on the premise that an exploit code is subject to several constraints in terms of the exploit code size and control flow. Subsequently, these constraints will help determine if a byte stream is data or program-like code.

Exploit Code Size. For every vulnerable buffer, an attacker can potentially can write arbitrary amount of data past the bounds of the buffer, but this will most likely result in a crash as the writes may venture into unmapped or invalid memory. This is seldom the goal of a remote exploit and in order to be successful, the exploit code has to be carefully constructed to fit inside the buffer. Each vulnerable buffer has a limited size and this in turn puts limits on the size of the transmitted infection vector.

Branch Instructions. The interesting part of a branch instruction is the branch target and for an exploit code, the types of branch targets are limited - 1) due to the uncertainty involved during a remote infection, control flow cannot be transferred to any arbitrary memory location, 2) due to the size constraints, branch targets can be within the payload component and hence, calls/jumps beyond the size of the flow are meaningless, or 3) due to the goals which must be achieved, the exploit code must eventually transfer control to a system call. Branch instructions of interest [2] are `jmp` family, `call/ret` family, `loop` family and `int`.

System Calls. Even an attacker must look to the underlying system call subsystem to achieve any practical goal such as a privileged shell. System calls can be invoked either through the library interface (`glibc` for Linux and `kernel32.dll`,

`ntdll.dll` for Windows) or by directly issuing an interrupt. If the former is chosen, then we look for the preferred base load address for libraries which on Linux is 0x40—— and 0x77—— for Windows. Similarly, for the latter, then the corresponding interrupt numbers are `int 0x80` for Linux and `int 0x2e` for Windows.

A naive approach to exploit code detection would be to just look for branch instructions and their targets, and verify the above branch target conditions. However, this is not adequate due to the following reasons, necessitating additional analysis. First, in our experience, although the byte patterns satisfying the above conditions occur with only a small probability in a network flow, it is still not sufficiently small to avoid false positives. Second, the branch targets may not be obvious due to indirect addressing, that is, instead of the form 'call 0x12345678', we may have 'call eax' or 'call [eax]'.

There two general categories of exploit code from a static analysis viewpoint depending on the amount of information that can be recovered. To the first category belong those types of exploit code which are transmitted in plain view such as known exploits, zero-day exploits and metamorphic exploits. The second category contains exploit code which is minimally exposed but still contains some hint of control flow, and polymorphic code belongs to this category. Due to this fundamental difference, we approach the process of elimination for polymorphic exploit slightly differently although the basic methodology is still on static analysis. Note that if both polymorphism and metamorphism are used, then the former is the dominant obfuscation. We now turn to the details of our approach starting with binary disassembly.

4.2   Disassembly

In general, Intel disassembly is greedy in nature, quickly consuming bytes until the actual instruction stream is reached. As this happens regardless of where the disassembly begins, it is already an efficient instruction recovery mechanism. Convergent dissembly is also useful when data is embedded inside the instruction stream. As an illustration, consider the following byte sequence which begins with a `jmp` instruction and control flow is directed over a set of data bytes into NOPs. Observe that convergence holds good even in this case with the data bytes being interpreted as instructions, and although there is a loss of one NOP, it still synchronizes with the instruction stream.

Byte sequence: EB 04 DD FF 52 90 90

```
00000000: EB04    jmp short 0x6
00000002: DD0A    fisttp dword [edx]
00000004: DD      db 0xDD
00000005: FF5290  call near [edx-0x70]
00000008: 90      nop
```

However, there are caveats to relying entirely on convergence; the technique is lossy and this does not always bode well for static analysis because while the loss of instructions on the NOOP sled is not serious, loss of instructions inside the exploit code can be.

Due to the nature of conditions being enforced, branch instructions are important. It is desirable to recover as many branch instructions as possible, but it comes at the price of a large processing overhead. Therefore, depending on whether the emphasis is on efficiency or accuracy, two disassembly strategies arise.

Strategy I: (Efficiency). The approach here is to perform binary disassembly starting from the first byte without any additional processing. The convergence property will ensure that at least a majority of instructions including branch instructions has been recovered. However, this approach is not resilient to data injection.

Strategy II: (Accuracy). The network flow is scanned for opcodes corresponding to branch instructions and these instructions are recovered first. Full disassembly is then performed over the resulting smaller blocks. As a result, no branch instructions are lost.

The latter variation of binary disassembly is slower not only because of an additional pass over the network flow but also the number of potential basic blocks that may be identified. The resulting overheads could be significant depending on the network flow content. For example, one can expect large overheads for network flows carrying ASCII text such as HTTP traffic because several conditional branch instructions are also printable characters, such as the 't' and 'u', which binary disassembly will interpret as `je` and `jne` respectively.

4.3   Control and Data Flow Analysis

Our control and data flow analysis is a variation of the standard approach. Having performed binary disassembly using one of the aforementioned strategies, we construct the control flow graph (CFG). Basic blocks are identified as usual via block leaders - the first instruction is a block leader, the target of a branch instruction is a block leader, and the instruction following a branch instruction is also a block leader. A basic block is essentially a sequence of instructions in which flow of control enters at the first instruction and leaves via the last. For each block leader, its basic block consists of the leader and all statements upto but not including the next block leader. We associate one of three states with each basic block - *valid*, if the branch instruction at the end of the block has a valid branch target, *invalid*, if the branch target is invalid, and *unknown*, if the branch target is not obvious. This information helps in pruning the CFG. Each node in the CFG is a basic block, and each directed edge indicates a potential control flow. We ignore control predicate information, that is, `true` or `false` on outgoing edges of a conditional branch. However, for each basic block tagged as invalid, all incoming and outgoing edges are removed, because that block cannot appear in any execution path. Also, for any block, if there is only one outgoing edge and that edge is incident on an invalid block, then that block is also deemed invalid. Once all blocks have been processed, we have the required CFG. Figure 4 shows the partial view of a CFG instance. In a typical CFG, invalid blocks form

a very large majority and they are excluded from any further analysis. The role of control flow analysis in our approach is not only to generate the control flow graph but also to greatly reduce the problem size for static analysis. The remaining blocks in a CFG may form one or more disjoint chains (or subgraphs), each in turn consisting of one or more blocks. In Figure 4, blocks numbered 1 and 5 are invalid, block 2 is valid and ends in a valid library call, and blocks 3 and 4 form a chain but the branch instruction target in block 4 is not obvious. Note that the CFG does not have a unique entry and exit node, and we analyze each chain separately.
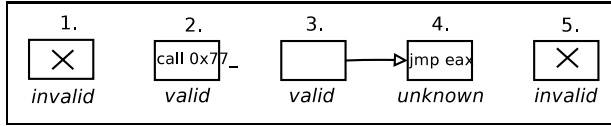

Fig. 4. A snapshot of a typical CFG after control flow analysis

We use data flow analysis based on program slicing to complete the process of elimination. Program slicing is a decomposition technique which extracts only parts of a program relevant to a specific computation, and there is a rich literature on this topic [34,20,14]. For our purpose, we adopt the backward static slicing technique approach proposed by Weiser [28], who used the control flow graph as an intermediate representation for his slicing algorithm. This algorithm has a running time complexity of $O(v \times n \times e)$, where v, n, e are the numbers of variables, vertices and edges in the CFG, respectively. Given that there are only a fixed number of registers on Intel platform, and that the number of vertices and edges in a typical CFG is almost the same, the running time is $O(n^2)$. Other approaches exist which use different representations such as program dependence graph (PDG) and system dependence graph (SDG), and perform graph reachability based analysis [30,17]. However, these algorithms incur additional representation overheads and are more relevant when accuracy is paramount.

In general, a few properties are true of any chain in the reduced CFG. Every block which is not the last block in the chain has a branch target which is an offset into the network flow and points to its successor block. For the last block in a chain, the following cases capture the nature of the branch instruction.

Case I: Obvious Library Call. If the last instruction in a chain ends in a branch instruction, specifically `call`/`jmp`, but with an obvious target (immediate/absolute addressing), then that target must be a library call address. Any other valid branch instruction with an immediate branch target would appear earlier in the chain and points to the next valid block. The corresponding chain can be executed only if the stack is in a consistent state before the library call, hence, we expect `push` instructions before the last branch instruction. We compute a program slice with the slicing criterion $< s, v >$, where s is the statement number of the `push` instruction and v is its operand. We expect v to be defined before it is used in the instruction. If these conditions are satisfied, then an alert

is flagged. Also, the byte sequences corresponding to the last branch instruction and the program slice are converted to a signature (described later).

Case II: Obvious Interrupt. This is other case of the branch instruction with an obvious branch target, and the branch target must be a valid interrupt number. In other words, the register `eax` is set to a meaningful value before the interrupt. Working backwards from the `int` instruction, we search for the first use of the `eax` register, and compute a slice at that point. If the `eax` register is assigned a value between 0-255, then again an alert is raised, and the appropriate signature is generated.

Case III: The `ret` Instruction. This instruction alters control flow but depending on the stack state. Therefore, we expect to find at some point earlier in the chain either a `call` instruction, which creates a stack frame or instructions which explicitly set the stack state (such as `push` family) before `ret` is called. Otherwise, executing a `ret` instruction is likely to cause a crash rather than a successful exploit.

Case IV: Hidden Branch Target. If the branch target is hidden due to register addressing, then it is sufficient to ensure that the constraints over branch targets presented in 4.1 hold over the corresponding hidden branch target. In this case, we simply compute a slice with the aim of ascertaining whether the operand is being assigned a valid branch target. If so, we generate alert.

Polymorphic Exploit Code. As mentioned earlier, polymorphic exploit code is handled slightly differently. Since only the decryptor body can be expected to be visible and is often implemented as a loop, we look for evidence of a cycle in the reduced CFG, which can be achieved in $O(n)$, where n is the total number of statements in the valid chains. Again, depending on the addressing mode used, the loop itself can be obvious or hidden. For the former case, we ascertain that at least one register being used inside the loop body has been initialized outside the body. An alternative check is to verify that at least one register inside the loop body references the network flow itself. If the loop is not obvious due to indirect addressing, then the situation is similar to case IV. We expect that the branch target to be assigned a value such that control flow points back to the network flow. By combining this set of conditions with the earlier cases, we have a generic exploit code detection technique which is able to handle both metamorphic and polymorphic code.

Potential for Evasion. Any static analysis based approach has a limitation in terms of the coverage that can be achieved. This holds true even for our approach and an adversary may be able to synthesize which evades our detection technique. However, there are some factors in our favor. Obfuscations typically incur space overheads and the size of the vulnerable buffer is a limiting factor. Moreover, in the reduced CFG, we scan *every* valid chain and while it may be possible to evade detection in a few chains, we believe it is difficult to evade detection in all of them. Finally, the above rules for detection are only the initial set and

may require updating with time, but very infrequently as compared to current signature-based systems.

4.4   Signature Generation

Control flow analysis produces a pruned CFG and data flow analysis identifies interesting instructions within valid blocks. A signature is generated based on the bytes corresponding to these instructions. Note that we do not convert a whole block in the CFG into a signature because noise from binary disassembly can misrepresent the exploit code and make the signature useless. The main consideration while generating signatures is that while control and data flow analysis may look at instructions in a different light, the signature must contain the bytes in the order of occurrence in a network flow. We use the regular expression representation containing wildcards for signatures since the relevant instructions and the corresponding byte sequences may be occur disconnected in the network flow. Both Bro and Snort (starting from version 2.1.0) support regular expression based rules, hence, our approach makes for a suitable signature generation engine.

5   An Exploit Detection Sensor

So far we have described the inner workings of our exploit detection algorithm. We now turn to its application in the form of a network flow-based exploit detection sensor called *styx*. Figure 5 presents a design overview. There are four main components: flow monitor, content sieve, malicious program analyzer and executable code recognizer. The executable code recognizer forms the core component of styx, and other components assist it in achieving its functionality and improving detection accuracy.
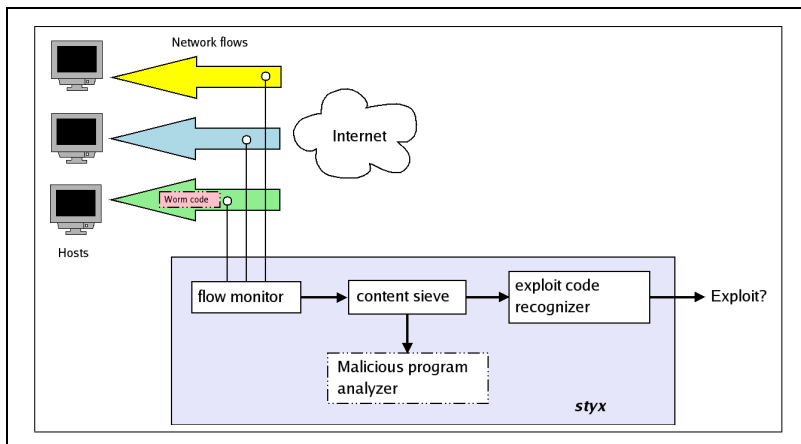
Fig. 5. Architecture of an exploit code detection sensor based on convergent static analysis

Flow Monitor. Our view of the information transfer over networks is that of network flows. The main task of the *flow monitor* is to intercept incoming/outgoing packets and reconstruct the corresponding flow data. Network flows can be unidirectional or bidirectional, and both directions can contain executable worm code. We implemented the flow monitor using `tcpflow`, which captures and reassembles the network packets. We have used `tcpflow` mainly because it is an off-the-shelf open-source tool which is readily available and can be easily modified to suit our requirements. `tcpflow` writes all the information exchanged between any two hosts into two separate files, one for each direction.

We consider both TCP and UDP flows. Reconstruction of TCP flows is fairly straightforward even when packets arrive out of order. UDP is an unreliable protocol and when packets arrive out of order, reconstructing the intended network stream is not possible. In such cases, styx will likely miss the embedded exploit code. However, this is not such a serious issue as it may seem because if the UDP packets arrived in a different order than what a exploit code author intended, then it is unlikely that infection will be successful. This is perhaps why not many exploit code which transmit using UDP, and when such worms are implemented, the worm code is very small. For example, the Slammer/Sapphire worm used UDP and was small enough to fit in only one UDP packet.

Content Sieve. Some network flows may contain programs which can pass all our tests of exploit code detection leading to false positives. It is therefore necessary to make an additional distinction between program-like code and programs. The *content sieve* is responsible for filtering content which may interfere with the executable code recognizer component. To this end, before deploying styx, it is necessary to specify which services may or may not contain executable code. This information is represented as a 3-tuple (π, τ, υ), where π is the standard port number of a service, τ is the type of the network flow content, which can be data-only (denoted by `d`) or data-and-executable (denoted by `dx`), and υ is the direction of the flow, which is either incoming (denoted by `i`) or outgoing (denoted by `o`). For example, (`ftp, d, i`) indicates an incoming flow over the ftp port has data-only content type. Further fine-grained rules could be specified on a per-host basis. However, in our experience we have seen that for a large organization which contains several hundred hosts, the number of such tuples can be very large. This makes fine-grained specification undesirable more so because it puts a large burden on the system administrator rather than the impact it may have on styx's performance. If a rule is not specified, then data-only network flow content is assumed by default for the sake of convenience since most network flows carry data. Therefore, the content sieve is activated only when a flow has a rule indicating that it is not data-only.

The content sieve has been implemented to identify Linux and Microsoft Windows executable programs. Our data set shows that occurrence of programs inside flows is not very common and when they do occur, it can attributed to downloads of third-party software from the Internet. We believe that the occurrence of programs could be much higher in popular peer-to-peer file sharing networks. However, the security policy at the enterprise where the data was

collected, prevents use of such networks and therefore our data set is not representative of this scenario.

Programs on the Linux and Windows platform follow the ELF [7] and the PE [8] executable formats respectively. We describe the methodology for detecting an ELF executable; the process is similar for a PE executable. The network flow is scanned for the characters 'ELF' or equivalently, the consecutive bytes 454C46 (in hexadecimal). This byte sequence usually marks the start of a valid ELF executable. Next, we look for the following positive signs to ascertain that the occurrence of the these bytes was not merely a coincidence.

ELF Header: The ELF header contains information which describes the layout of the entire program, but for our purposes, we require only certain fields to perform sanity checks. For example, the `e_ident` field contains machine independent information and must assume meaningful values (see [7]), `e_machine` must be `EM_386`, `e_version` must be 1, etc. As our data set indicates, these checks are usually adequate. But if additional confirmation is required, then we also perform the next two checks.

Dynamic Segment: From the ELF header, we find the offset of the Program Header and subsequently, the offset of the Dynamic Segment. If this segment exists, then the executable uses dynamic linkage and the segment must contain the names of the external shared libraries such as `libc.so.6`.

Symbol and String Tables: Also from the ELF header, we find the offset of symbol and string tables. The string tables must strictly contain printable characters. Also, the symbol table entries must point to valid offsets into the string table.

The format of a PE header closely resembles an ELF header and we perform similar checks as described above. A Windows PE executable file [8] starts with a legacy DOS header, which contains two fields of interest - `e_magic`, which must be the characters 'MZ' or equivalently the bytes `5A4D` (in hexadecimal), and `e_lfanew`, which is the offset of the PE header.

It is highly unlikely that normal network data will conform to all these specifications. Therefore, when all of them are satisfied, it is reasonable to assume that an executable program has been found. As the next step, we mark the boundaries of the executable and exclude it from further analysis.

Malicious Program Analyzer. While the main aim of the content sieve is to identify full programs inside network flows which in turn contain executable code fragments so that they do not interfere with our static analysis scheme, there is a beneficial side-effect. Since we have the capability of locating programs inside network flows, they can passed as input to other techniques [24] or third-party applications such as anti-virus software. This also helps when an attacker transfers malicious programs and rootkits following a successful exploit. The *malicious program analyzer* is a wrapper encapsulating this functionality and is a value-added extension to expfl0w.

Executable Code Recognizer. After the preliminary pre-processing, the network flow is input to the *executable code recognizer*. This component primarily

implements the convergent static analysis approach described in Section 4. It is responsible both for raising alerts and generating the appropriate signatures.

6   Evaluation

We have performed experimental evaluation primarily to determine detection efficacy and performance overheads. The first dataset used in the experiments consisted of 17 GB of network traffic collected over a few weeks at a enterprise network, which is comprised mainly of Windows hosts and a few Linux boxes. The organization policy prevented us from performing any live experiments. Instead, the data collection was performed with only the flow monitor enabled, while algorithmic evaluation was performed later by passing this data through the rest of the exploit detection sensor in a quarantined environment. During the period this data was collected, there was no known worm activity and neither did any of the existing IDS sensors pick up exploit-based attacks. Therefore, this dataset is ideal to measure the false positive rates as well running times of our algorithm. In order to specifically measure detection rates, we used exploits generated using the Metasploit framework [3].

6.1   Detection

When performing detection against live network traffic, the exploit code detection sensor did not report the presence of exploit code in any of the network flows. The live traffic which was collected contained mostly HTTP flows and these flows had the potential to raise false positives due to the ASCII text and branch instruction problem mentioned earlier. However, since we use further control and data flow analysis, none of the CFGs survived the process of elimination to raise any alarms. The other types of network flows were either binary or encrypted and the reduced CFGs were far smaller in size and number, which were quickly discarded as well. However, we warn against hastily inferring that our approach has a zero false positive rate. This is not true in general because our technique is probabilistic in nature and although the probability of a false positive may be very small, it is still not zero. But this is already a significant result since one of the downsides of deploying an IDS is the high rate of false positives.

Next we studied detection efficacy and possible ways in which false negatives can occur. Using the Metasploit framework [3], it is possible to handcraft not only several types of exploit code but also targeted for different platforms. There are three main components in the Metasploit framework - a NOOP sled generator with support for metamorphism, a payload generator, and a payload encoder to encrypt the payload. Therefore, one can potentially generate hundreds of real exploit code versions. We are interested only in Intel-based exploits targeted for Windows and Linux platforms. We discuss the interesting representative test cases.

Metamorphic Exploit Code. Due to the nature of our detection process, the payload of metamorphic code is not very different from any other vanilla exploit

code. The Metasploit framework allows the generation of metamorphic NOOP sleds. The following is the relevant code segment which is the output of the 'Pex' NOOP sled generator combined with the 'Windows Bind Shell' payload. Note the different single-byte opcodes which form the NOOP sled. We have also shown portions of the payload which were a part of the first valid chain encountered when analyzing the flow containing the exploit code. The corresponding signature which was generated was: 60 .* E3 30 .* 61 C3. Note that stack frame which was created using `pusha` was popped off using `popa`, but just the mere presence of stack-based instructions in the chain is deemed adequate evidence.

```
00000001 56        push esi
00000002 97        xchg eax,edi
00000003 48        dec eax
00000004 47        inc edi
         . . .
00000009 60        pusha
0000000A 8B6C2424 mov ebp,[esp+0x24]
0000000E 8B453C   mov eax,[ebp+0x3c]
00000011 8B7C0578 mov edi,[ebp+eax+0x78]
         . . .
0000001F E330     jecxz 0x51
         . . .
00000051 61        popa
00000052 C3        ret
```

Polymorphic Exploit Code. We generated a polymorphic exploit code using the 'PEX encoder' over the 'Windows Bind Shell' payload. This encoder uses a simple XOR-based scheme with the key encoded directly in the instruction. We highlight the following segment of code, where `0xfd4cdb57` at offset 0000001F is the key. The encrypted payload starts at offset 0000002B. Our approach was able to detect this polymorphic code because of the conditions satisfied by the `loop` instruction with `esi` register being initialized before the loop and referenced within the loop. The corresponding signature which was generated was: 5E 81 76 0E 57 DB 4C FD 83 EE FC E2 F4. A caveat is that this signature is very specific to this exploit code instance due to the key being included in the signature. If the key is excluded then, we have a more generic signature for the decryptor body. However, this requires additional investigation and part of our future work.

```
00000018 E8FFFFFF        call 0x1C
0000001C FFC0            inc eax
0000001E 5E              pop esi
0000001F 81760E57DB4CFD  xor dword [esi+0xe],0xfd4cdb57
00000026 83EEFC          sub esi,byte -0x4
00000029 E2F4            loop 0x1F
0000002B C7              db 0xC7
```

Worm Code. We used Slammer/Sapphire as the test subject. The worm code follows a very simple construction and uses a tight instruction cycle. The whole worm code fits in one UDP packet. The payload used was an exploit against the MS SQL server. Again, both versions of our approach were able to detect the worm code and generated the signature: `B8 01 01 01 01 .* 50 E2 FD`, which corresponds to the following portion of the worm code [4]. This is the first executable segment which satisfies the process of elimination and our algorithm exits after raising an alert.

```
0000000D B801010101 mov eax,0x1010101
00000012 31C9       xor ecx,ecx
00000014 B118       mov cl,0x18
00000016 50         push eax
00000017 E2FD       loop 0x16
```

In our experience, both variations of our exploit code detection algorithm were equally effective in detecting the above exploit code versions. This was mainly because the payload consisted of continuous instruction streams. However, carefully placed data bytes can defeat the fast disassembly scheme, making the accurate scheme more relevant.

6.2   Performance Overheads

We compared our approach against a signature-based detection system - Snort. Several factors contribute to the runtime overheads in both approaches. For Snort, the overheads can be due to network packet reassembly, signature table lookups, network flow scanning and writing to log files. On the other hand, for our approach, overheads are be caused by network packet reassembly, binary disassembly and static analysis. We are mainly interested in understanding running-time behavior, and therefore, implemented and compared only the core detection algorithms. Moreover, since we conducted our experiments in an offline setting, all aspects of a complete implementation cannot be meaningfully measured.

The single most important factor is the network flow size. In order to correctly measure running time for this parameter only, we either eliminated or normalized other free parameters. For example, Snort's pattern matching algorithm also depends on the size of the signature repository while in our approach signatures are a non-factor. We normalized it by maintaining a constant Snort signature database of 100 signatures throughout the experiment. The bulk of these signatures were obtained from `http://www.snort.org` and the rest were synthesized. All experiments were performed on 2.6 GHz Pentium 4 with 1 GB RAM running Linux (Fedora Core 3).

Figure 6 shows the results obtained by running both variations of our approach against Snort's pattern matching. We considered four kinds of network flows based on flow content. As is evident from the plots, pattern matching is extremely fast and network flow size does not appear to be a significant factor. In contrast, the running time of our approach shows a non-negligible dependence on the size of network flows. Both variations of our approach display a linear relationship,
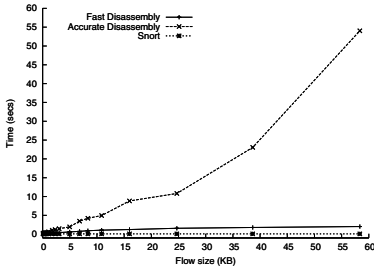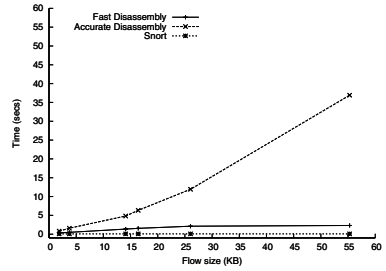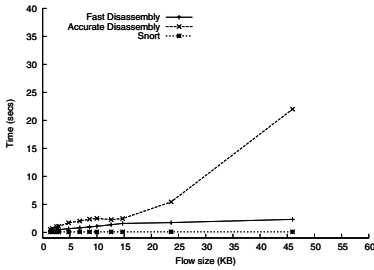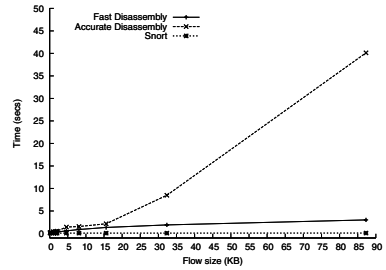
Fig. 6. Comparison of network flow processing times between our approach (both fast and accurate disassembly) and Snort's pattern matching

however, the slopes are drastically different. The fast disassembly version incurs far smaller overheads, while the accurate disassembly version may be impractical in the context of live network traffic when flow sizes are large. Referring again to pattern matching, We also believe that a larger signature repository is also not likely to affect running time significantly. However, the downside is that since detection requires the signature database to be constantly updated and maintained, there is a large space overhead which increases with each additional signature. Our approach scores over pattern matching in this regard since it does not require maintaining any such tables.

Deployment Issues. The runtime performance studies provide us with useful insight into practical deployment scenarios. Snort can be deployed at various points including a network tap configuration at the organization's network entry point where the volume of network is the highest. In contrast, our approach may not be very suitable at this point of deployment; even the faster version may show noticeable latency. Instead, internal routers or end hosts are more practical deployment sites. There is yet another possibility. Since the input to the core algorithm is eventually a stream of bytes, our approach, sans the network processing components, can be implemented directly into programs for additional validation of all incoming program inputs at runtime.

Improvements. In our performance measurements experiments, as expected, HTTP traffic incurred the highest overheads because of the printable ASCII characters being more frequent than other flows, which resulted in a larger number of branch instructions and basic blocks. For example, a typical flow of 10 KB in size returned 388 basic blocks for the fast version and 1246 basic blocks for the accurate version. This number can be reduced by preprocessing a network flow and removing application level protocol headers containing ASCII text. Since most traffic is HTTP, this may be a worthwhile improvement. Other general improvements can be made by optimizing the implementation. Another distinct possibility is to implement our approach in hardware since it has no dynamic components such as a signature repository. We believe this can lead to very significant performance improvements.

7   Conclusion and Future Work

In this paper, we have described an efficient static analysis based litmus test to determine if a network flow contains exploit code. This is a significant departure from existing content-based detection paradigms. Our approach has the ability to detect several different types of exploit code without any maintenance costs, making for a true plug-n-play security device. On the downside, although our static analysis technique is very efficient compared to traditional static analysis approaches, it is still not fast enough to handle very large network traffic, and therefore, there are deployment constraints. Therefore, we believe our approach cannot replace existing techniques altogether, but rather work in tandem with them.

There are three main avenues which we are actively pursuing as a part of our ongoing and future work. First, we are investigating ways to sensitize our static analysis based detection against potential obfuscations. This will greatly improve the long-term relevance of our approach rather than being a stop-gap solution. Second, we are studying possible ways in which our approach can be sped up significantly. This would close the performance gap between signature-based detection schemes and our technique. Finally, after satisfactory maturation, we will perform more exhaustive testing in a live deployment setting.

References

1. F-secure virus descriptions : Santy. `http://www.fsecure.com/v-descs/santy_a.shtml`.
2. *IA-32 Intel Architecture Software Developer's Manual.*
3. *Metasploit Project.* http://www.metasploit.com/.
4. Slammer/Sapphire Code Disassembly. `http://www.immunitysec.com/downloads/disassembly.txt`.
5. *The Twenty Most Critical Internet Security Vulnerabilities (Updated) The Experts Consensus.* `http://files.sans.org/top20.pdf`.
6. VX heavens. `http://vx.netlux.org`.

7. *Tool Interface Standard (TIS), Executable and Linking Format (ELF) Specification, Version 1.2*, 1995.

8. *Microsoft Portable Executable and Common Object File Format Specification, Revision 6.0*, 1999. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx.

9. C. Cifuentes and K. Gough. Decompilation of Binary Programs. *Software Practice & Experience*, 25(7):811–829, July 1995.

10. M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium (Security'03)*, pages 169–186. USENIX Association, USENIX Association, aug 2003.

11. M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium (Security '03)*, 2003.

12. C. Cifuentes and M. V. Emmerik. UQBT: Adaptable binary translation at low cost. *Computer*, 33(3):60–66, 2000.

13. C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security Symposium*, San Antonio, TX, January 1998.

14. D.W. Binkley and K.B. Gallagher. Program Slicing. *Advances in Computers*, 43:1–50, 1996.

15. H.H. Feng, J.T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, pages 194–, 2004.

16. S. Hittel. Detection of jump-based ids-evasive noop sleds using snort, May 2002. http://aris.securityfocus.com/rules/020527-Analysis-Jump-NOOP.pdf.

17. S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.

18. R. Jones and P. Kelly. Bounds Checking for C. http://www-ala.doc.ic.ac.uk/phjk/BoundsChecking.html.

19. J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy*, May 2004.

20. M. Kamkar. An overview and comparative classification of program slicing techniques. *J. Syst. Softw.*, 31(3):197–214, 1995.

21. H.-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium (Security '04)*, 2004.

22. O. Kolesnikov, D. Dagon, and W. Lee. Advanced polymorphic worms: Evading ids by blending in with normal traffic. Technical Report GIT-CC-04-15, College of Computing, Georgia Institute of Technology, 2004.

23. N. Krawetz. The Honeynet files: Anti-honeypot technology. *IEEE Security and Privacy*, 2(1):76–79, Jan-Feb 2004.

24. C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security 2004 (Security '04)*, 2004.

25. W. Landi. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

26. C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static analysis. In *10th ACM Conference of Computer and Communications Security (CCS)*, 2003.

27. LURHQ Threat Intelligence Group. Phatbot trojan analysis. `http://www.lurhq.com/phatbot.html`

28. M. Weiser. *Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, The University of Michigan, Ann Arbor, Michigan, 1979.

29. D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.

30. K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184, 1984.

31. A. Pasupulati, J. Coit, K. Levitt, S. Wu, S. Li, R. Kuo, and K. Fan. Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. In *9th IEEE/IFIP Network Operation and Management Symposium (NOMS 2004) to appear*, Seoul, S. Korea, May 2004.

32. G. Ramalingam. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994.

33. S. Staniford, V. Paxson, and N. Weaver. How to 0wn the internet in your spare time, 2002.

34. F. Tip. A survey of program slicing techniques. Technical Report CS-R9438, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1994.

35. T. Toth and C. Krügel. Accurate buffer overflow detection via abstract payload execution. In *Recent Advances In Intrusion Detection (RAID)*, pages 274–291, 2002.

36. J. Twycross and M.M. Williamson. Implementing and testing a virus throttle. In *Proceedings of the 12th Usenix Security Symposium (Security '03)*, 2003.

37. D. Wagner and D. Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the IEEE Symposium on Security and Privacy*, page 156. IEEE Computer Society, 2001.

38. K. Wang and S. J. Stolfo. Anomalous payload-based network intrusion detection. In *Recent Advances In Intrusion Detection (RAID)*, pages 203–222, 2004.

39. N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A taxonomy of computer worms. In *First ACM Workshop on Rapid Malcode (WORM)*, 2003.

40. N. Weaver, S. Staniford, and V. Paxson. Very fast containment of scanning worms. In *USENIX Security Symposium*, pages 29–44, 2004.

41. N. Weaver, S. Staniford, and V. Paxson. Very fast containment of scanning worms. In *USENIX Security Symposium*, pages 29–44, 2004.

42. C.C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and early warning for internet worms. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 190–199. ACM Press, 2003.

43. C.C. Zou, W. Gong, and D. Towsley. Code red worm propagation modeling and analysis. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 138–147. ACM Press, 2002.