A

Seminar report

On

Parasitic Computing

Submitted in partial fulfillment of the requirement for the award of degree
Of ECE

SUBMITTED TO:
www.studymafia.org

SUBMITTED BY:
www.studymafia.org

Preface

I have made this report file on the topic **Parasitic Computing**; I have tried my best to elucidate all the relevant detail to the topic to be included in the report. While in the beginning I have tried to give a general view about this topic.

My efforts and wholehearted co-corporation of each and everyone has ended on a successful note. I express my sincere gratitude to ……..who assisting me throughout the preparation of this topic. I thank him for providing me the reinforcement, confidence and most importantly the track for the topic whenever I needed it.

Acknowledgement

I would like to thank respected Mr…….. and Mr. ……..for giving me such a wonderful opportunity to expand my knowledge for my own branch and giving me guidelines to present a seminar report. It helped me a lot to realize of what we study for.

Secondly, I would like to thank my parents who patiently helped me as i went through my work and helped to modify and eliminate some of the irrelevant or un-necessary stuffs.

Thirdly, I would like to thank my friends who helped me to make my work more organized and well-stacked till the end.

Next, I would thank Microsoft for developing such a wonderful tool like MS Word. It helped my work a lot to remain error-free.

Last but clearly not the least, I would thank The Almighty for giving me strength to complete my report on time.

CONTENTS

INTRODUCTION

The net is a fertile place where new ideas/products surface quite often. We have already come across many innovative ideas such as Peer-to-Peer file sharing, distributed computing etc. Parasitic computing is a new in this category. Reliable communication on the Internet is guaranteed by a standard set of protocols, used by all computers. The Notre Dame computer scientist showed that these protocols could be exploited to compute with the communication infrastructure, transforming the Internet into a distributed computer in which servers unwittingly perform computation on behalf of a remote node.

In this model, known as “parasitic computing”, one machine forces target computers to solve a piece of a complex computational problem merely by engaging them in standard communication. Consequently, the target computers are unaware that they have performed computation for the benefit of a commanding node. As experimental evidence of the principle of parasitic computing, the scientists harnessed the power of several web servers across the globe, which–unknown to them–work together to solve an NP complete problem.

Sending a message through the Internet is a sophisticated process regulated by layers of complex protocols. For example, when a user selects a URL (uniform resource locator), requesting a web page, the browser opens a transmission control protocol (TCP) connection to a web server. It then issues a hyper-text transmission protocol (HTTP) request over the TCP connection. The TCP message is carried via the Internet protocol (IP), which might break the message into several packages, which navigate independently through

numerous routers between source and destination. When an HTTP request reaches its target web server, a response is returned via the same TCP connection to the user's browser. The original message is reconstructed through a series of consecutive steps, involving IP and TCP; it is finally interpreted at the HTTP level, eliciting the appropriate response (such as sending the requested web page). Thus, even a seemingly simple request for a web page involves a significant amount of computation in the network and at the computers at the end points.

In essence, a `parasitic computer' is a realization of an abstract machine for a distributed computer that is built upon standard Internet communication protocols. We use a parasitic computer to solve the well known NP-complete satisfiability problem, by engaging various web servers physically located in North America, Europe, and Asia, each of which unknowingly participated in the experiment. Like the SETI@home project, parasitic computing decomposes a complex problem into computations that can be evaluated independently and solved by computers connected to the Internet; unlike the SETI project, however, it does so without the knowledge of the participating servers. Unlike `cracking' (breaking into a computer) or computer viruses, however, parasitic computing does not compromise the security of the targeted servers, and accesses only those parts of the servers that have been made explicitly available for Internet communication.

THE NP-COMPLETE PROBLEM

A problem is assigned to the NP (nondeterministic polynomial time) class if it is verifiable in polynomial time by a Nondeterministic Turing Machine (A nondeterministic Turing Machine is a "parallel" Turing Machine which can take many computational paths simultaneously, with the restriction that the parallel Turing machines cannot communicate.). A problem is NP-hard if an algorithm for solving it can be translated into one for solving any other NP-problem. NP-hard therefore means "at least as hard as any NP-problem", although it might, in fact, be harder. A problem which is both NP and NP-hard is said to be an NP-Complete problem. Examples of NP-Complete problems are the traveling salesman problem and the satisfiability problem.

The `satisfiability' (or SAT) problem involves finding a solution to a Boolean equation that satisfies a number of logical clauses. For example, (x1 XOR x2) AND (x2 AND x3) in principle has 23 potential solutions, but it is satisfied only by the solution x1 = 1, x2 = 0, and x3 = 1. This is called a 2-SAT problem because each clause, shown in parentheses, involves two variables. The more difficult 3-SAT problem is known to be NP complete, which in practice means that there is no known polynomial-time algorithm which solves it. Indeed, the best known algorithm for an n-variable SAT problem scales exponentially with n . Here we follow a brute-force approach by instructing target computers to evaluate, in a distributed fashion, each of the 2^n potential solutions.

Travelling salesmen problem involves working out the shortest route that a fictional salesman would have to take to visit all possible locations on a hypothetical map. The more locations on the hypothetical map means more potential routes, and the longer it would take any single computer to crank through all possible combinations. But by sharing the job of working out which route is shortest, the total time it takes to solve any particular travelling salesman problem can be vastly reduced.

THEORY

To solve many NP complete problems, such as the traveling salesman or the satisfiability problem, a common technique is to generate a large number of candidate solutions and then test the candidates for their adequacy. Because the candidate solutions can be tested in parallel, an effective computer architecture for these problems is one that supports simultaneous evaluation of many tests.

Four Notre Dame professors recently discovered a new Internet vulnerability that is commonly known as "parasitic computing." The researchers found a way to "trick" Web servers around the world into solving logic math problems without the server's permission. The researchers found that they could tag a logic problem onto the check sum (the bit amount that is sent when a Web page is requested) and the Web server would process the request. When a Web page was requested without the correct check sum, the server would not respond to the request.

Each of the math problems that were tagged on to the request by the researchers was broken down into smaller pieces that were evaluated by servers in North America, Europe and Asia. The results from each were used to build a solution. Using a remote server, the team divided the problem into packages, each associated with a potential answer. The bits were then hidden inside components of the standard transmission control protocol of the Internet, and sent on their merry way.

The major discovery in this experiment is that other computers are answering logical questions without knowledge of doing so. The work is performed without consent, creating an ethical dilemma. The technique does not violate the security of the unknowing server; it only uses areas that are open for public access. They find it useful because they found a way to use a computer elsewhere to solve a problem.

Here, the computer consists of a collection of target nodes connected to a network, where each of the target nodes contains an arithmetic and logic unit (ALU) that is capable of performing the desired test and a network interface (NIF) that allows the node to send and receive messages across the network. A single home parasite node initiates the computation, sends messages to the targets directing them to perform the tests, and tabulates the results.

Owing to the many layers of computation involved in receiving and interpreting a message, there are several Internet protocols that, in principle, could be exploited to perform a parasitic computation. For example, an IP-level interpretation could force routers to solve the problem, but such an implementation creates unwanted local bottlenecks. To truly delegate the computation to a remote target computer, we need to implement it at the TCP or higher levels. Potential candidate protocols include TCP, HTTP, or encryption/ decryption with secure socket layer (SSL).

HOW TO TRICK OTHER PEOPLE'S COMPUTERS TO SOLVE A MATH PROBLEM?

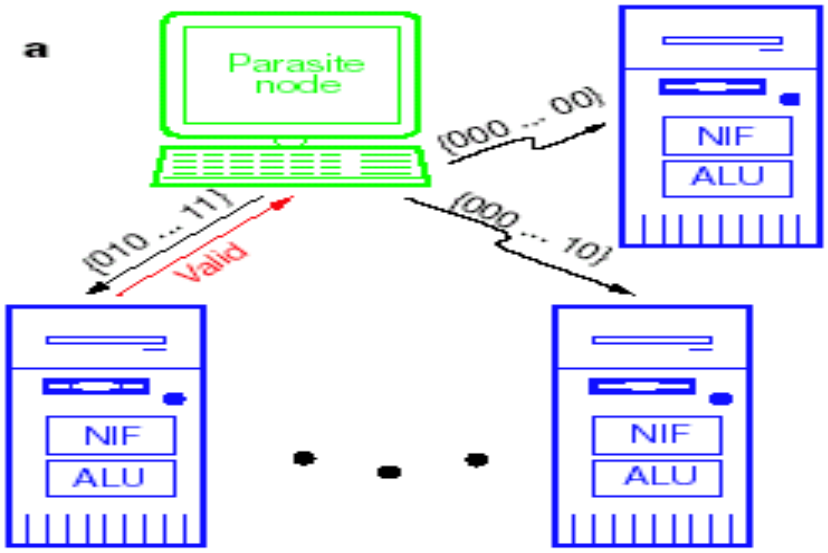


Figure 1: Schematic diagram of our prototype parasitic computer. A single parasite node coordinates the computations occurring remotely in the Internet protocols. It sends specially constructed messages to some number of targeted nodes , which are web servers consisting of an arithmetic and logic unit (ALU) and a network interface (NIF).ie. a single home parasite node initiates the computation, sends messages to the targets directing them to perform the tests, and tabulates the results

The communication system that brings you the Web page of your choice can be exploited to perform computations. In effect, one computer can co-opt other Internet computers to solve pieces of a complex computational problem. The enslaved computers simply handle what appear to be routine Web page requests and related messages, but the disguised messages ingeniously encode

possible solutions to a mathematical problem. If the solution is correct, a message returns to

the original sender. The target computers are unaware that they have performed computation for the benefit of a commanding node

Key component:

The seemingly simple request for a Web page involves a significant amount of frenetic behind-the-scenes computation aimed at finding, delivering, and displaying the desired page. One key component, governed by the so-called transmission control protocol (TCP), involves a calculation to determine whether a chunk of data was delivered without error-CHECKSUM COMPUTATION. Information sent across the Internet is typically split into small chunks, or packets, that travel—often independently of each other—to their common destination. Each packet bears a header providing data about its source and destination and carrying a numerical value related to the packet's contents. When a computer receives a packet of information, it checks for errors by performing a calculation and comparing the result with the numerical value in the packet's header (see "How TCP error detection works," below). Such a calculation would detect, for example, the change of one bit from 0 to 1 or 1 to 0. Packets found to be corrupted are discarded.

COMPUTING WITH TCP

The implementation of parasitic computing exploits a reliability mechanism in the transmission control protocol (TCP). During package transfer across the Internet, messages can be corrupted, that is, the bits can change. TCP contains a checksum that provides some data integrity of the message. To achieve this, the sender computes a checksum and transmits that with the message. The receiver also computes a checksum, and if it does not agree with the sender's, then the message was corrupted The sender of a TCP segment computes a checksum over the entire segment, which provides reliability against bit errors that might occur in transport.

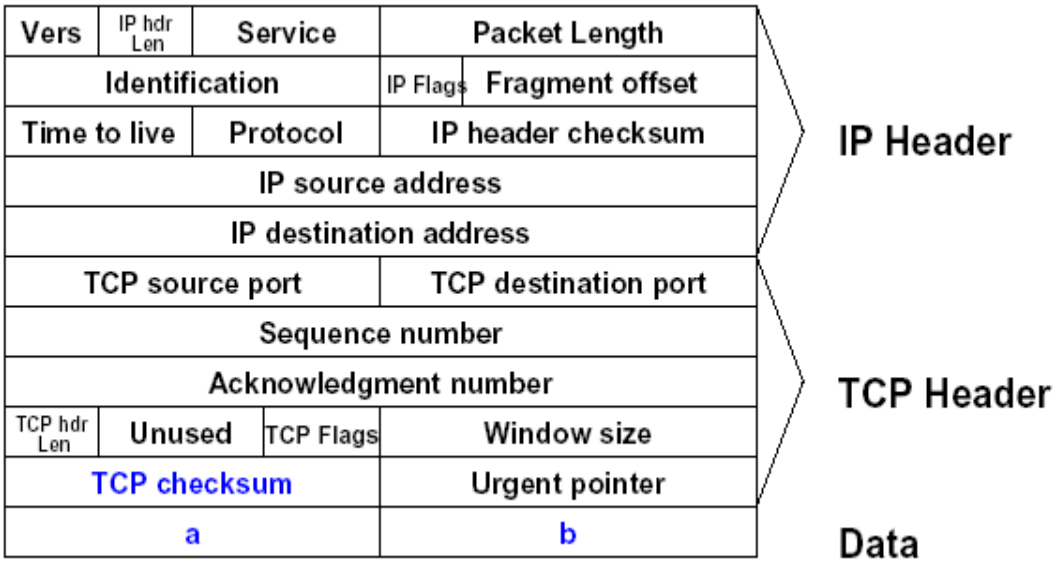


Figure 2: TCP/IP segment used for parasitic computing

It inserts the complement of the checksum in the message. The receiver also computes a checksum in order to verify data integrity. The receiver drops an entire segment if its checksum does not add up, assuming

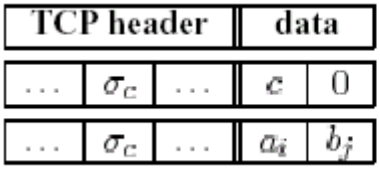
that the message was corrupted in transit. Because TCP is reliable, a sender will retransmit each segment until it is acknowledged by the receiver. One property of the TCP checksum

function is that it forms a sufficient logical basis for implementing any Boolean logic function, and by extension, any arithmetic operation3.To implement a parasitic computer using the checksum function we need to design a special message that coerces a target server into performing the desired computation. As a test problem we choose to solve the well known 'satisfiability' (or SAT) problem, which is a common test for many unusual computation methods.

The TCP checksum is exploited to answer the following question:
Is a+b equal to c?

The checksum value in a TCP packet is determined by c . The data contains 16-bit words a and b . TCP computes the checksum, if a+b != c , then TCP rejects the segment. Therefore, a message is a "valid" TCP segment if and only if a+b = c .

Suppose one needs to find two numbers which add up to a certain value, a+b =c . One could generate guesses for a and b , add them, and test if the sum is equal to c . The checksum mechanism of TCP can be used to solve this problem. First, compute a checksum for the answer, c , as the data part of the the TCP segment. Next, send a TCP segment where the data part contains candidate addends a_i and b_i, as shown below.



Dummy packet for computing checksum

Solution packet

Finally, continue to send TCP segments until one is received—meaning that the checksum was verified. For any a_i+b_j != c , the TCP segment is dropped because the checksum verification fails. Consequently, only the correct solution (where a+b=c) is a "well-formed" TCP segment. The checksum in the TCP header σ_c not c because the checksum computation is computed over the entire header and it is complemented. If the header fields are the same, the checksum computed for a packet with a and b as data is the same as that

computed for a packet with c and 0 (pad to keep length the same). figure 2 shows a schematic diagram of the specially constructed TCP segment that is used in our exploit. The first 20 bytes are the IP header, the next 20 are the TCP header, and the last 4 bytes are the data. The values of a and b and the checksum ( ) are shown in blue.

The parasite node creates 2n specially constructed messages designed to evaluate a potential solution. The design of the message, exploiting the TCP checksum, is described in Fig. 2. These messages are sent to many target servers throughout the Internet. Note that while we choose the simpler 2-SAT case to illustrate the principle behind the coding, the method can be extended to code the NP complete 3-SAT problem as well, as explained in the Supplementary Information. The message received by a target server contains an IP header, a TCP header, and a candidate solution (values for xi). The operators in the Boolean equation determine the value of the checksum, which is in the TCP header. The parasite node injects each message into the network at the IP level (Fig. 1), bypassing TCP. After receiving the message, the target server verifies the data integrity of the TCP segment by calculating a TCP checksum. The construction of the message (Fig. 3) ensures that the TCP checksum fails for all messages containing an invalid solution to the posed SAT problem. Thus, a message that passes the TCP checksum contains a

correct solution. The target server will respond to each message it receives (even if it does not understand the request). As a result, all messages containing invalid solutions are dropped in the TCP layer. Only a message which encodes a valid solution 'reaches' the target server, which sends a response to the 'request' it received.

We have implemented the above scheme using as a parasitic node an ordinary desktop machine with TCP/IP networking. The targeted computers are various web servers physically located in North America, Europe, and Asia, each of which unwittingly participated in the experiment. As explained earlier, our parasite node distributed 2n messages between the targets. Because only messages containing valid solutions to the SAT problem pass through TCP, the target web server received only valid solutions. This is interpreted as an HTTP request, but it is of course meaningless in this context. As required by HTTP, the target web

server sends a response to the parasitic node, indicating that it did not understand the request. The parasite node interprets this response as attesting to the validity of the solution. As expected and by design, incorrect solutions do not generate responses for the web server. A typical message sent by the parasite, and a typical response from a target web server are included in the Supplementary Information. Our technique does not receive a positive acknowledgement that a solution is invalid because an invalid solution is dropped by TCP. Consequently, there is a possibility of false negatives: cases in which a correct solution is not returned, which can occur for two reasons. First, the IP packet could be dropped, which might be due to data corruption or congestion. Normally TCP provides a reliability mechanism against such events, but our current implementation cannot take advantage of this. Second, because this technique exploits the TCP checksum, it circumvents the function the checksum provides. The TCP checksum catches errors that are not caught

in the checks provided by the transport layer, such as errors in intermediate routers and the end points.

Measurements show that the TCP checksum fails in about 1 in 210 messages. The actual number of TCP checksum failures depends on the communication path, message data, and other factors. To test the reliability of our scheme, we repeatedly sent the correct solution to several host computers located on three continents. The rate of false negatives with our system ranged from 1 in about 100 to less than 1 in 17,000. The implementation offered above represents only a proof of concept of parasitic computing. As such, our solution merely serves to illustrate the idea behind parasitic computing, and it is not efficient for practical purposes in its current form. Indeed, the TCP checksum provides a series of additions and a comparison at the cost of hundreds of machine cycles to send and receive messages, which makes it computationally inefficient. To make the model viable, the computation-to-communication ratio must increase until the computation exported by the parasitic node is larger than the amount of cycles required by the node to solve the problem itself instead of sending it to the target. However, we emphasize that these are drawbacks of the presented implementation and do not represent fundamental obstacles for parasitic computing. It remains to be seen,

however, whether a high-level implementation of a parasitic computer, perhaps exploiting HTTP or encryption/decryption could execute in an efficient manner.

It is important to note that parasitic computing is conceptually related but philosophically different for cluster computing, which links computers such that their cumulative power offers computational environments comparable to the best supercomputers. A prominent example of cluster computing is the SETI program, which has so far enlisted over 2.9 million

computers to analyse radio signals in search of extraterrestrial intelligence. In cluster computing, the computer's owner willingly downloads and executes software, which turns his computer into a node of a vast distributed computer. Thus, a crucial requirement of all cluster computing models is the cooperation of the computer's owner. This is also one of its main limitations, as only a tiny fraction of computer owners choose to participate in such computations. In this respect, parasitic computing represents an ethically challenging alternative for cluster computing, as it uses resources without the consent of the computer's owner. Although parasitic computing does not compromise the security of the target, it could delay the services the target computer normally performs, which would be similar to a denial-of-service attack, disrupting Internet service. Thus, parasitic computing raises interesting ethical and legal questions regarding the use of a remote host without consent, challenging us to think about the ownership of resources made available on the Internet. Because parasitic computation exploits basic Internet protocols, it is technically impossible to stop a user from launching it. For example, changing or disrupting the functions that are exploited by parasitic computing would simply eliminate the target's ability to communicate with the rest of the Internet. In summary, parasitic computing moves computation onto what is logically the communication infrastructure of the Internet, blurring the distinction between computing and communication. We have shown that the current Internet infrastructure permits one computer to instruct other computers to perform computational tasks that are beyond the target's immediate scope. Enabling all computers to swap information and services they are needed could lead to unparalleled emergent behaviour, drastically altering the current use of the Internet.

CHECKSUM COMPUTATION:

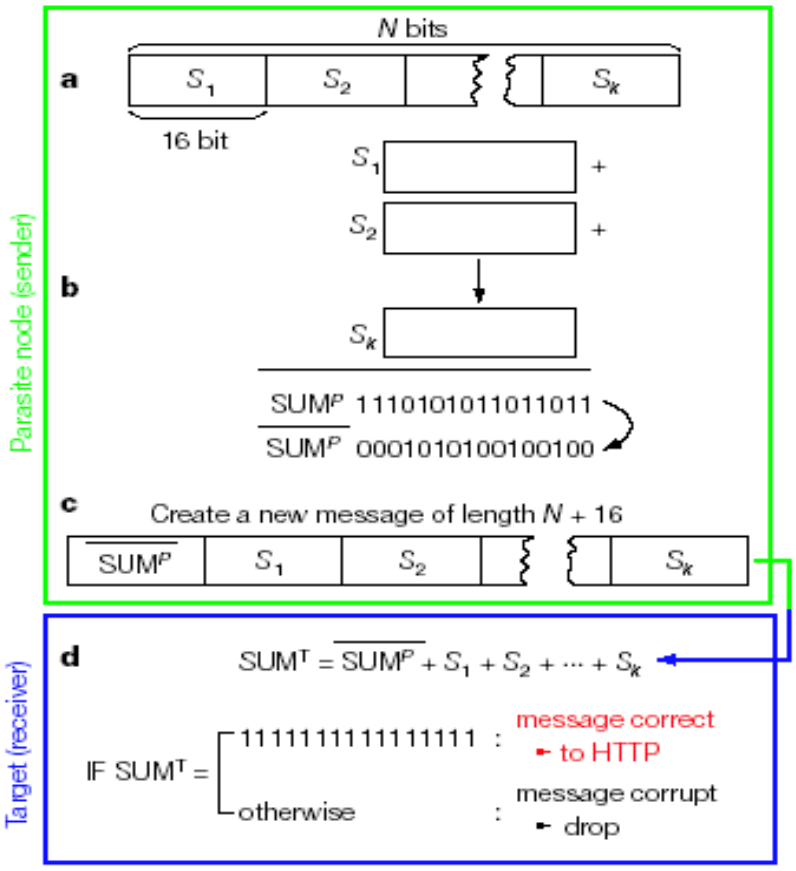


Figure 6

Figure 6 shows the TCP checksum. The checksum is a simple function performed by all web servers (in TCP), which allows a recipient to check if the received message has been corrupted during transmission. The sender (parasitic node) breaks the message consisting of N bits into 16-bit words, shown as S_1, S_2,……, S_k. The k words are added together using binary one's-complement arithmetic, providing the sum denoted as SUM^P . Next, the sender performs a bit-wise complement on the checksum, so that every bit is flipped: a 0 becomes a 1 and a 1 becomes a 0, obtaining $\overline{SUM^P}$. An example of a checksum and its complement are shown in the figure 6b. The sender incorporates the complement of the checksum into the header of the message

as shown in 6c. The receiving computer (target) again breaks the received message into 16-bit segments and adds them together. The value of the checksum SUM^T calculated by the target is $\overline{SUM^P} + SUM^P$, the first term coming from the header and the second term being the contribution from $S_1 + S_2 + \ldots S_k$. As SUM^P and $\overline{SUM^P}$ are complementary, the checksum obtained by the receiver has to be 1111111111111111. If any bit along the message has been corrupted during transmission, the checksum obtained by the target will be different from all ones, in which case the target drops the message. A non-corrupted message is passed to the HTTP protocol, which will attempt to interpret its content.

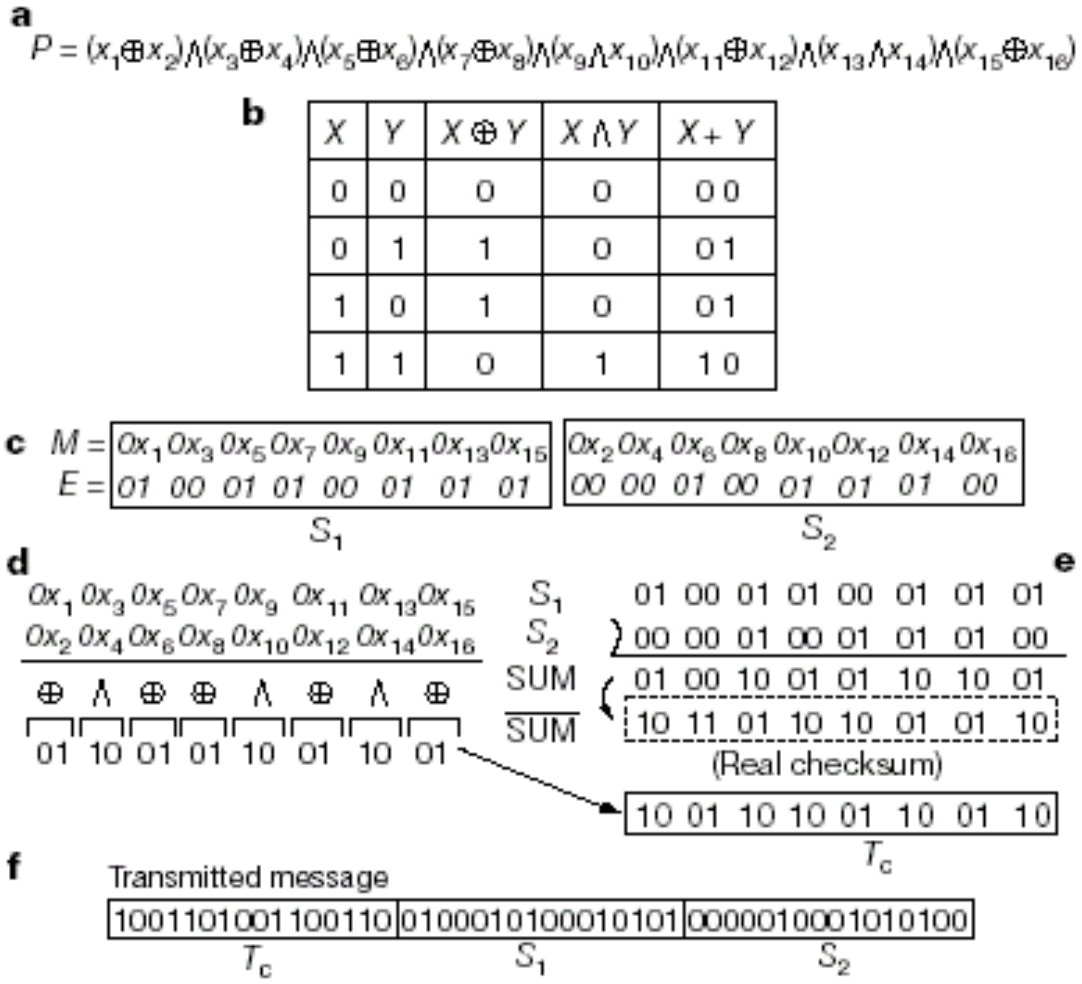


Figure 7.

Figure 7 shows how satisfiability is decided using checksum. The 2-SAT problem shown in fig 7a involves 16 variables {x 1; x 2;…..; x 16} and the operators AND (^) and XOR ($\oplus$). The logical table of XOR, AND and the binary sum (+) is shown in fig 7b. In order to get a TRUE answer for P, each clause shown in separate parentheses in fig7a needs to be independently TRUE. To evaluate the value of P we generate a 32-bit message M that contains all 16 variables, each preceded by a zero. As an illustration, we show a possible solution E. TCP groups the bits of the received message in two 16-bit segment and adds them together . As shown in fig 7d, this will result in adding each (x_i, x_{i+1}) pair together where i is odd. The sum can have four outcomes. Comparing the sum with the ($\oplus$) column in the table in 7b, we notice that a TRUE answer for the XOR clause ($x_i \oplus x_{i+1}$) coincides with the (01) result of the (x_i, x_{i+1}) sum. Similarly, if the clause has an AND operator, (x_i ^ x_{i+1}) is true only when the checksum is (10). This implies that for a set of variables {x1, x2,……,x16} that satisfies P the checksum will be determined by the corresponding operators only (that is, a $\oplus$ should give (01) for the sum check, and for ^ the sum is (10)). For illustration, in fig 7d we show the formal lineup of the variables, while in fig 7e we show an explicit example. The correct complemented checksum for E should be $\overline{\text{SUM}}$ = 10110110100110. In contrast, the parasitic computer places in the header the transmitted checksum T_c = 1001101001100110, which is uniquely determined by the operators in P. To turn the package into a parasitic message the parasitic node prepares a package, shown in fig 7f, preceded by a checksum T_c, and continued by a 32-bit sequence (S1, S2), which represent one of the 2^{16} potential solutions. If S1 and S2 do not represent the correct solution, then the checksum evaluated by the target TCP will not give the correct sum (111……11). The TCP layer at the target concludes that the message has been corrupted, and drops it. However, if S1 and S2 contain the valid solution, the message is sent

to HTTP. The web server interprets the solution as an HTTP request; however, because it is not a valid HTTP request, the web server responds with a message saying something like `page not found' . Thus, every message to which the parasite node receives a response is a solution to the posed SAT problem .

ALGORITHM

An 8-variable 2-SAT problem problem is shown below.

$$(x_1 \oplus x_2) \wedge (x_3 \wedge x_4) \wedge (x_1 \oplus \neg x_4) \wedge (x_2 \oplus x_5) \wedge (x_3 \wedge x_6) \wedge (x_4 \wedge x_7) \wedge (x_6 \oplus x_8) \wedge (x_2 \oplus x_4) \quad (1)$$

The solution vector, $\vec{x}$ } _ , has 8 elements that can range over 0 and 1. Thus there are 2^8 possible solutions for $\vec{x}$_ . The only correct solution to (1) is $\vec{x}$ =[1.0.1,1,1,1,1,0]^T Our algorithm tests each solution using a specially constructed TCP/IP packet.

The high-level algorithm is:

S = **create** TCP segment

S.checksum = checksum

Foreach $\vec{x}$

S.data = pad with zeroes($\vec{x}$} _ s)

send S

receive answer

if answer = **true** . **write** $\vec{x}$ is a solution

First, we create a TCP segment that contains all the standard header information required by the protocol. Next the checksum field is set. The solution determines the checksum, therefore, there is a single checksum for all tests. In the main loop of the algorithm, a test solution is placed into the data field of the segment. Then the packet is sent, and we wait for an answer. A

correct solution induces a response from the remote node. Therefore, a response means a correct solution. An incorrect solution is deduced by not receiving a response. This is done by timing out: if a response is not received within a certain amount of time it is presumed a negative answer. This is discussed in greater detail.

IMPLEMENTATION

In our implementation a single master node controls the execution of the algorithm. There are several ways to implement the basic algorithm discussed above. The two major choices are (a) concurrency and (b) connection reuse. Regarding (a), the master node can have many computations occurring in the web concurrently. Each concurrent computation requires a separate TCP connection to a HTTP host. Regarding (b), before a TCP connection can be used, it must be established. Once established, TCP segments can be sent to the remote host. When multiple guesses are sent in one connection, it is impossible to know to which guess a correct solution refers to. For example, suppose guess $<b_1 , c_1 >$ and $< b_2 , c_2>$ are sent one after the other in a single connection. Further suppose that only one solution is correct. We expect to get one response back. But we cannot tell to which solution the response refers.

The implementation used in this paper is a prototype that is not designed for efficiency of execution. In our prototype implement there is no concurrency and each connection is used for exactly one computation.

RELIABLE COMMUNICATIONS

Any message can get lost. In a reliable system, the sender of a message saves a copy of the message and waits for an acknowledgement of the message. If after some time, the sender has not received an acknowledgement, it will re-send the message (from the copy). The sender will continue to do this until an acknowledgement is received.

In general, there is no upper bound on how long a message might take to be delivered. Consequently, in a distributed system, it is not possible to distinguish between a lost message and a delayed message. Therefore, a message is assumed lost after some *time-out* period. A time-out value that is too small declares too many delayed messages as lost. On the other hand, a value that is too large unnecessarily slows down the system.

Our exploit circumvents the reliability mechanism in TCP. Furthermore, because an invalid solution fails the checksum, it is as if it never arrived. Therefore, the receiver will not send an acknowledgement of the message. There are two undesirable outcomes that could occur:

- A false negative occurs when a packet for a valid solution is dropped due to a data corruption or congestion.
- A false positive occurs when a bit error changes an invalid result into an valid result

The latter is very rare statistically and all but impossible in practice. Although the former is also unusual, it is frequent enough that it should be considered further.

First, let's consider the errors that are caught by the TCP checksum. Every transmission link (hardware devices such as ethernet) computes a checksum on its packets. The TCP checksum catches errors that pass the link checksum, but still have some data corruption. Because the data was not damaged in transmission (where it would have been caught by the transmission link checksum), it must have occurred in an intermediate system

(router, bridge, gateway, etc.) or at an end point (sender or receiver) [3]. Such errors occur very infrequently.

Research shows that the TCP checksum fails about 1 in 2^{20} messages [4]. The probability of receiving a false positive is the probability of a error times the probability that it changes an invalid solution into a valid solution. The probability of the latter event is infinitesimal.

Second, an IP packet might be dropped due to data corruption or congestion. The ordinary TCP reliability mechanism handles this, but it is disabled in our prototype. Our test show false negatives occur between 1 in about 100 and less than 1 in 17,000. The error rate is strongly correlated with the distance (number of hops) between end points.

DEALING WITH AN UNRELIABLE SYSTEM

This section describes two approaches to using this unreliable system. First, one could ask every question multiple times. The probability of false negatives is almost certainly uncorrelated. Therefore, if P is the probability of a false negative, then P^n is the probability of n false negative, because $P << 1$ the likelihood of a false negative all but disappears for small values of n.

Second, one could ask a question, Q, and its complement !Q. Absent any errors, one will get exactly one response. If no response is received, one must assume that a problem occurred. Then, the questions should be asked again. This solution results in a reliable system, but requires the every question also have a complement.

IMPLEMENTING THE 3-SAT PROBLEM

In the checksum, one can use up to three variables without overflow, as $1+1+1=11_2$. Thus a 3- SAT problem that has 3 variables per clause can be encoded in a way similar to the 2-SAT implementation described before, assuming that there are appropriate operators whose logical tables match the checksum. The algorithm described before does not have to be modified. The only change is how the packet is constructed. Each candidate solution has contains three 16-bit words, whish are added together and compared to answer the question: is a+b+c equal to d?

The sender computes the checksum over three 16-bit words and the header, as shown below.

TCP header			data			
...	σ_d	...	d	0	0	Dummy packet for computing checksum
...	σ_d	...	a_i	b_j	c_j	Solution packet

The data part of the packet contains three data words. Data words are constructed with zero padding, as done in the 2-SAT problem. On the receiver side, a checksum in computed over the TCP packet just as before. A response is sent only if a+b+c=d does indeed equal. The only difference is between this and the 2-SAT problem, is that the packet is 2 bytes longer.

FEATURES OF PARASITIC COMPUTING

- parasitic computing theoretically offers the chance to use the vast computational power of the whole Internet.
- Several large computational problem can be solved by engaging various web servers physically located in different parts of the world, each of which unknowingly participated in the experiment.
- ethically challenging alternative for cluster computing, as it uses resources without the consent of the computer's owner.
- parasitic computing does not compromise the security of the targeted servers, and accesses only those parts of the servers that have been made explicitly available for Internet communication.

DISADVANTAGES

- **Communication to computation ratio:**

Currently, Internet-wide parasitic problem solving is only a theoretical consideration, because the method employed by Mr Barabási and his colleagues is "computationally inefficient", as they admit. They had to invest hundreds of machine cycles to send and receive messages until they finally achieved the solution to their mathematical problem. To make parasitic computing really useful, the ratio between the invested communication and the resulting computation has to be dramatically improved. The Notre Dame researchers suggest that exploiting HTTP or encryption/decryption could solve the efficiency problem.

- **Delayed Services**

The legal aspects of parasitic computing are far from clear. The security of the target computer is in no way compromised. It is just simple communication, using only areas specifically earmarked for public access. But it could slow machines down by engaging them in a computational conversation, which would be similar to the disruption of Internet services by a denial-of-service attack. And there is only one way to prevent the parasites from sucking computational power out of your computer – disconnecting it.

CONCLUSION

Parasitic computing moves computation onto what is logically the communication infrastructure of the Internet, blurring the distinction between computing and communication. The Notre Dame scientists have shown that the current Internet infrastructure permits one computer to instruct other computers to perform computational tasks that are beyond the target's immediate scope. Enabling all computers to swap information and services they are needed could lead to unparalleled emergent behavior, drastically altering the current use of the Internet.

The implementation offered above represents only a proof of concept of parasitic computing. As such, the solution merely serves to illustrate the idea behind parasitic computing, and it is not efficient for practical purposes in its current form.

Indeed, the TCP checksum provides a series of additions and a comparison at the cost of hundreds of machine cycles to send and receive messages, which makes it computationally inefficient. To make the model viable, the computation-to-communication ratio must increase until the computation exported by the parasitic node is larger than the amount of cycles required by the node to solve the problem itself instead of sending it to the target.

However, these are drawbacks of the presented implementation and do not represent fundamental obstacles for parasitic computing. It remains to be seen, however, whether a high-level implementation of a parasitic computer, perhaps exploiting HTTP or encryption/ decryption could execute in an efficient manner.

REFERENCES

- www.google.com
- www.wikipedia.com
- www.studymafia.org