Supplementary Material

(for Parasitic Computing, by Albert-László Barabási, et al.)

1 Encoding 3-SAT problem using TCP checksum

In the checksum, one can use up to three variables without overflow, as $1 + 1 + 1 = 11_2$. Thus a 3-SAT problem can be encoded in a way similar to the 2-SAT implementation described in the manuscript, assuming that there are appropriate operators whose logical tables match the checksum. There are 3 different operations in addition to the NOT operator for each variable that can be performed by the TCP checksum. (Note: that the solution to the 2-SAT problem uses AND and XOR between variables and NOT operator.)

Operation I: ALL TRUE (AND)—corresponding to $\mathrm{SUM} = x + y + z = 3 = 11_2$

$$(x \wedge y \wedge z) = \mathrm{TRUE\ iff\ all\ of\ three\ variables\ are\ true}$$

Operation II: TWO TRUE—corresponding to $\mathrm{SUM} = x + y + z = 2 = 10_2$

$$\begin{aligned}(x : y : z) &= \mathrm{TRUE\ iff\ only\ two\ of\ three\ variables\ are\ true} \\ &= (\neg x \wedge y \wedge z) \vee (x \wedge \neg y \wedge z) \vee (x \wedge y \wedge \neg z) \\ &= (\neg x \vee \neg y \vee \neg z) \wedge (x \vee y) \wedge (y \vee z) \wedge (z \vee x)\end{aligned}$$

Operation III: ONE TRUE—corresponding to $\mathrm{SUM} = x + y + z = 1 = 01_2$

$$\begin{aligned}(x.y.z) &= \mathrm{TRUE\ iff\ only\ one\ of\ three\ variables\ is\ true} \\ &= (x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge y \wedge \neg z) \vee (\neg x \wedge \neg y \wedge z) \\ &= (x \vee y \vee z) \wedge (\neg x \vee \neg y) \wedge (\neg y \vee \neg z) \wedge (\neg z \vee \neg x)\end{aligned}$$

For operations II and III, (x:y:z) and (x.y.z), the first definition looks simple, however, because we are using Conjunctive Normal Form (connected with AND($\wedge$) between clauses) it is more appropriate to use the second definition. Although a 2-SAT problem is not NP-complete, a mixed problem, where any number (greater than zero) of 3-SAT clauses is combined with 2-SAT clauses, is NP-complete.

To illustrate the use of these operators, consider the following sentence with eight variables $x_1, x_2, ..., x_8$:

$$(x_1.x_2.x_3) \wedge (\neg x_1 : x_4 : \neg x_5) \wedge (\neg x_2 \wedge \neg x_3 \wedge x_6) \wedge (x_4.\neg x_6.x_7) \wedge$$
$$(x_2.x_5.\neg x_7) \wedge (x_3 : \neg x_5 : x_8) \wedge (x_2 : x_6 : x_8) \wedge (\neg x_1.x_3.x_6)$$

The only solution is $(1, 0, 0, 1, 0, 1, 0, 1)$. The checksum that would force TCP to drop everything but the correct solution has the form 0110 1101 0110 1001. As a test we have implemented their 3-SAT problems as well as described in §4.3.

2 Computing with TCP

Our implementation of parasitic computing exploits a reliability mechanism in the transmission control protocol (TCP). The sender of a TCP segment computes a checksum over the entire segment, which provides

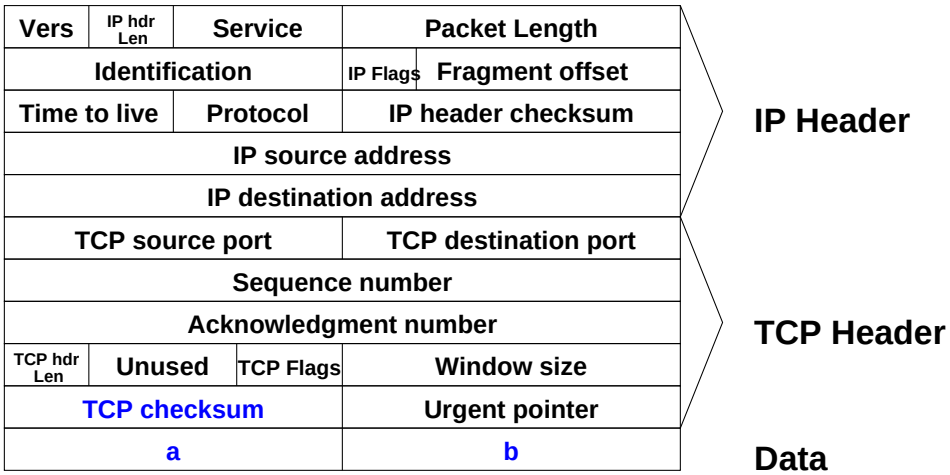

Figure 1: TCP/IP segment used for parasitic computing

reliability against bit errors that might occur in transport. 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 [1, 2].

We exploit the TCP checksum 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 $a + b \neq c$, 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_j, as shown below.

TCP header			data		
...	σ_c	...	c	0	Dummy packet for computing checksum
...	σ_c	...	a_i	b_j	Solution packet

Finally, continue to send TCP segments until one is received—meaning that the checksum was verified. For any $a_i + b_j \neq c$, 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 is 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 1 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

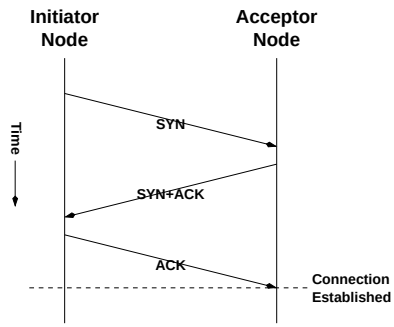

Figure 2: Establishing a TCP connection.

values of a and b and the checksum (σ_c) are shown in blue.

TCP is a connection-oriented protocol—meaning that a connection must first be established before data is sent. A connection is established with three messages, see Figure 2. First, the initiator sends a SYN (the TCP SYN flag is set) to a remote node. If the remote node is willing to accept a new connection, it responds with a SYN/ACK (both the SYN and ACK flags are set) message. Finally, the initiator sends an ACK message [1, 2]. In these messages nodes exchange initial sequence numbers. The test TCP segments, constructed as above, are sent following the establishment of a connection.

Because we are connecting to an HTTP server, the response received depends on the configuration of the server. However, all we care about is whether there is a response or not. The correct solution results in a valid TCP segment, but it is not a valid HTTP request. Therefore, the response will be something like that in Figure 3.

3 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, as shown in Figure 1.

The high-level algorithm is:

S = **create** TCP segment
S.checksum = checksum
foreach $\vec{x_i} \rightarrow$
  S.data = pad_with_zeroes($\vec{x_i}$)
  send S
  receive answer
  if answer = **true** $\rightarrow$ **write** $\vec{x_i}$ 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, as discussed in Figure 3 in the text;

```
<html>
<head>
<Title> Notre Dame Computer Science and Engineering </Title>
</head>
<body bgcolor=white>

<center>
<img align=middle src=http://www.nd.edu/NDGrafix/NDCSE.gif
alt="[ U_N_I_V_E_R_S_I_T_Y___o_f___N_O_T_R_E___D_A_M_E ]">
<h1></h1><h2><p> That feature is not implelemented on this server (501): <br> /index.html

<H2> If you feel this message is in error, please contact <br>
<em><a href="mailto:www@www.cse.nd.edu">www@www.cse.nd.edu</a>
</em></h2>
<H4> Please include the full URL that you are trying to access,
<br>or we may not be able to provide you with any assistance.<br><p>
<img align=bottom src=http://www.nd.edu/NDGrafix/NDBarThin.gif>

<p> <a href=http://www.cse.nd.edu/>
<img align=top src=http://www.nd.edu/NDGrafix/NDCSEHome.gif
alt="[BACK TO ND CSE HOME PAGE]"></a>
</center> </body></html>
```

Figure 3: Response from HTTP server

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 in §2 and §4.

4 Implementations

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.

4.1 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, and
- 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^{10} 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.

4.2 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 $p \ll 1$ the likelihood of a false negative all but disappears for small values of n.

Second, one could ask a question, $\mathcal{Q}$, and its complement, $\neg\mathcal{Q}$. 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.

4.3 Implementing the 3-SAT Problem

The 3-SAT problem has three variables per clause. This easily can be supported with the prototype. The algorithm described in §3 does not have to be modified. The only change is to how the packet is constructed. Each candidate solution contains three 16-bit words, which 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**			
$\ldots$	σ_d	$\ldots$	d	0	0	Dummy packet for computing checksum
$\ldots$	σ_d	$\ldots$	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 $a + b + c$ does indeed equal d. The only difference is between this and the 2-SAT problem, is that the packet is 2 bytes longer.

5 Scalability

Definition of 3-SAT problem: 3-SAT is a special case of the satisfiability problem whereby all formulae are in a special form. A *literal* is a Boolean variable or a negated Boolean variable, as in x or $\neg x$. A clause is several literals connected with ORs, as in $(x_1 \vee \neg x_2 \vee \neg x_3)$. A Boolean formula is in conjunctive normal form, called a CNF-formula, if all clauses are connected with ANDs, as in

$$(x_1 \vee \neg x_2 \vee \neg x_3 \vee x_4) \wedge (x_3 \vee \neg x_5 \vee x_6) \wedge (x_4 \vee x_5 \vee \neg x_6).$$

It is a 3CNF-formula if all the clauses have exactly three literals, as in

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_5 \vee x_6) \wedge (x_3 \vee \neg x_6 \vee x_4).$$

The 3-SAT problem is determining the satisfiability of a 3CNF-formula. It is possible to reduce any $(k > 3)$-SAT problem into a 3-SAT problem. First, if a formula has mixed ANDs and ORs, by the distributive laws,

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z),$$

$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z),$$

we can replace an OR of ANDs with an equivalent AND of ORs. Doing so may significantly increase the size of each sub-formula, but only by constant factor. By this process, we have written the formula in CNF. All that remains is to make all clauses contain three literals. In each clause that currently has one or two literals, replicate one literal until the total number is three. In each clause that has more than three literals, split it into several clauses and add additional variables to preserve the satisfiability or non-satisfiability of the original. For example, the following clause $(a_1 \vee a_2 \vee a_3 \vee a_4)$ is replaced with $(a_1 \vee a_2 \vee z) \wedge (\neg z \vee a_3 \vee a_4)$ where z is a new variable. It is straightforward to extend this into the clause containing n literals. For example, $(a_1 \vee a_2 \vee ... \vee a_n)$ can be rewritten with the $n - 2$ clauses $(a_1 \vee a_2 \vee z_1) \wedge (\neg z_1 \vee a_3 \vee z_2) \wedge (\neg z_2 \vee a_4 \vee z_3) \wedge ... \wedge (\neg z_{l-3} \vee a_{l-1} \vee a_l)$ Therefore, it is always possible to reduce $(k > 3)$-SAT problem into 3-SAT problem.

References

[1] Stevens, W. R. 1994. *TCP/IP Illustrated.* Addison-Wesley, Reading, Massachusetts.

[2] Forouzan, B. A. 2000. *TCP/IP Protocol Suite.* McGraw-Hill, Boston, Massachusetts.

[3] Jonathan Stone and Craig Partridge. When the CRC and TCP checksum disagree. In *SIGCOMM 2000*, September 2000.

[4] Jonathan Stone, Michael Greenwald, Craig Partridge, and Jim Hughes. Performance of checksums and CRCs over real data. *IEEE Trans. on Networks*, October 1998.