Automatic extraction of symbolic network signatures from malware code

Arjun Gurumurthy, Kausik Subramanian

Abstract

Malware is one of the primary threats to modern-day networks. While the most prevalent form of malware detection is endhost-based (e.g., matching code hashes), a detection mechanism integrated directly into the network offers a much-needed additional layer of protection. Towards this goal, we propose automatic generation of accurate network signatures of malware using program analysis techniques and SMT solvers. Informally, a network signature is the set of messages received by the malware to initiate attacks. These network signatures can be used by intrusion detection systems to detect malware traffic in the network. We present some preliminary work towards this goal.

1 Motivation

Malware is software designed to inflict illegitimate action and affect data, hosts or networks. These are used to disrupt computer operations, extract sensitive information, and gain access to private systems. Infected hosts can participate in distributed denial-of-service attacks (DDoS), and these class of attacks are very hard to detect and mitigate, leading to immense loss of revenues. A new shift is towards creating malware for profit called botnets, in which a customer can buy a network of infected hosts for a price. In such a setting, the malware is specifically designed to evade detection, thus motivating the need for better detection tools for malware.

The most prevalent form of malware detection is endhost-based detection (commonly known as anti-virus software). The detected malware's code is hashed and the code signatures are uploaded to a central repository. By maintaining an up-to-date repository of malware signatures, the endhost software can detect malware by comparing program signatures with the repository. However, endhost-based malware detection can fail due to various reasons [1]:

- **Anti-virus software is not operational or up to date.** Due to this, the endhost machine can be infected by malware without any noticeable effects, thus the user will be oblivious of the infection.
- **Packer programs.** To render programs stealthy, malware authors employ packer programs [6]. Packers change the program content so that its signature differs but its functionality has not changed, thus avoiding detection by endhost signature-based software.

Instead, a intrusion-detection mechanism integrated directly into the service providers network offers a much-needed additional layer of protection. While the malware code is different, the protocol of the malware used to talk to its control server (which issues orders to the malware) will not vary. Thus, by generating accurate *network signature* of the malware (which can informally defined as the set of messages sent and received by the malware to initiate an action), we can use the intrusion detection system in the network (for e.g., Bro [7]) to search for these particular signatures, and can be used to infer the presence of the malware in the network. However, manually inspecting malware code is an intractable approach (signatures may need to be extracted in real-time). Thus, we propose *automatic extraction of accurate network signatures from malware code using different static analysis techniques.*

2 Challenges

We abstract the problem to the following: given a program point, can we automatically provide an ordered set of symbolic packets that must be received (and sent) by the client such that the program reaches the program point. The symbolic packets can be used as network signatures and used to detect malware (a set of packets satisfying the symbolic packets in order will infer that there is a malware with high confidence). By constructing symbolic packet signatures, we can generalize the intrusion policy and reduce true negatives (thus not affecting legitimate traffic).

Malware code can be thought of as a program reacting based on input received packets received (possibly multi-threaded), which could require changes to existing reachability analyses tools like Codesurfer [8]. To extract symbolic network signatures, we need to define the abstract semantics of the program suited for network signatures, and define the *combine* and *extend* operators used for interprocedural analysis.

3 Malware program model

The malware program listens to incoming packets, and the network signature is the set of incoming packets which triggers an attack by the program. We propose a reasonable abstraction of the code of malware programs.

We model a malware program as a infinite reactive system of the form:

```
while(True) {
        packet = socket.receive();
        ...
}
```

Each iteration of the loop, the program receives a packet, and based on current state and packet received, it performs certain actions. Using static analysis techniques, we can identify two set of variables, *state* and *packet* variables. State variables represent the program state in terms of the protocol, which packet variables are used to represent the current packet being processed by the program.

```
while(True) {
        packet = socket.receive();
        ...
        if (a == 1 and packet[8] = "b") {
                a = a + 1;
        }
        ...
}
```

In the above code snippet, a is a state variable and *packet* is a packet variable. This also illustrates the packet processing function of the program, which based on the current input packet and state variables, either transitions to a new program state or performs an action (attack or send packets). The above code snippet represents the complete model of the malware program, where based on input packets and current state, the program transitions to a new state.

```
a = 0;
while(True) {
packet = socket.receive();
          if (a == 0 and packet[0] == ”a”) {
                    a = a + 1;
          }
          if (a == 1 and packet[0] == ”b”) {
                    a = a + 1;
          }
          if (a == 2 and packet[0] == ”c”) {
                    attack()  [the attack point is provided as
                                        input to analysis]
          }
}
```

4 Related Work

An important assumption that we make is that of the simplification of the botnet client code to a set of **if (state, packet)** statements. Looking at existing research, significant work has gone into automatic protocol reverse-engineering of botnet C&C protocols using dynamic binary analysis. This generally follows a two step process: Extract the protocol grammar that captures the structure of messages comprising the protocol. Knowing the protocol message format would enable the construction of the protocol state machine, which represents the sequence of messages as a specification of protocol states and transition between states (based on messages received).

Polyglot [3] is a system that automatically extracts the protocol message format using dynamic binary analysis. The intuition behind the design of the system is that the way that an implementation of the protocol processes the received application data reveals a wealth of information about the protocol.

Dispatcher [2] attempts to perform protocol reverse-engineering for active botnet infiltration. Essentially, it is a tool that makes use of Polyglot to extract the message format and in addition infers the field semantics for messages sent and received by the application. Thus, having knowledge of the network packets received would enable us to make inferences on how they could transition to an ’attack point’.

Prospex [4] is a system for protocol specification extraction for stateful network protocols - essentially, it reverse engineers the protocol state machine by clustering similar network messages and tracks protocol state changes. By merging similar states, a minimal state machine is produced.

All three systems use dynamic analysis techniques - mainly taint propagation. However, this would mean the code would need to execute in a sandboxed environment or be a part of a honeypot system. Static analysis techniques would be a better approach.

Our work focuses on static analysis of the malware code, without having to execute it or capture network traces.

5 Live variable analysis

We obtain network signatures using packet and state variables, the complexity increasing with respect to the number of state variables. To reduce the number of state variables, we perform live variable analysis for the program up until the attack point - that is, live variable analysis is done assuming the program terminates after the attack point. If the attack point is within one or more loops, then the assumed termination point would be immediately after all the loops are closed. For our use case, we can treat the entire code till the assumed termination point as a single block. Any variables that are not live within this block are ignored, and only the remaining live variables are used as state variables for the signature extraction step. Identifying live variables boils down to solving the dataflow equations:

$$in[n] = use[n] \quad \bigcup \quad (out[n] - def[n])$$

$$out[n] = \bigcup_{S \,\epsilon\, succ[n]} in[s]$$

where *in* and *out* represent the live variables entering and leaving the edge in the control flow graph. *def* and *use* represent the variables defined and referenced in an expression represented by a node in the CFG.

6 Signature Extraction

We now describe a procedure to generate a network signature of length n (length of the signature is the number of packets received to trigger attack) for a boolean malware program. We use the SMT solver Z3 [5] to find valid network signatures. In this section, we describe the constraints generated for the SMT solver from the program such that the model generated by the solver can be used to extract signatures.

6.1 Preliminaries

For a malware program M, let S denote the vector of boolean state variables in the program, and P denote the vector of boolean packet variables in the program[1]. Let the initial values of the state variables be S_0.

[1] For example, a 60 bit packet is a vector of 60 boolean variables.

As described in the earlier section, the malware packet processing code is a set of `if` statements whose predicates are over the set of state and packet variables. Let the set of `if` statements be denoted by I, and let us denote each `if` statement by $(\gamma, \delta) \in I$, where $\gamma(S,P)$ denotes the predicate on S and P, and $\delta(S,P)$ denotes the action on S if $\gamma(S,P)$ is satisfied. Therefore, the semantics of the `if` statement is as follows:

$$\gamma(S,P) \Rightarrow S' == \delta(S,P) \qquad (1)$$

where S' denotes the new state after the `if` block is executed.

Formally, given M, we need to find a sequence of packets p_0, $p_1, \ldots p_{n-1}$ such that M starting from initial state reaches the attack point when it receives the packets in that particular order. This sequence is packets is defined as a valid network signature of the malware program.

6.2 Transition Model

We denote the state vector before receiving the i^{th} packet as S_i. When packet i of the signature is received, mlaware M, based on S_i and P_i, will execute a `if` (γ_j, δ_j) block. For a given action block comprising assignment statements, the logical formula corresponding to the assignments is: $\delta(S,P) = s'_1 \Leftrightarrow \delta_1(S,P) \wedge s'_2 \Leftrightarrow \delta_2(S,P) \Leftrightarrow \wedge\ s'_{|S|} = \delta_{|S|}(S,P)$, where δ_i corresponds to the right hand side expression for the assignment statement $s_i = exp$. If the action has no assignment for a state variable s_i, $\delta_i = s_i$ (the value must remain unchanged, so we need to add this clause).

We can model the malware state transition by the following constraints:

$\forall i. 0 \leq i \leq n-2.\ \forall(\gamma,\delta) \in I.$

$$\gamma(S_i, P_i) \Rightarrow S_{i+1} \Leftrightarrow \delta(S_i, P_i) \qquad (2)$$

To find minimal signatures, every packet received by the malware must trigger a state change, which translated to the constraint that atleast one of the if predicates must be satisfied:

$$\forall i.0 \leq i \leq n-2. \bigvee_{(\gamma,\delta)\in I} \gamma(S_i, P_i) \tag{3}$$

Let the attack point block be (γ_M, δ_M). Therefore, a network signature of length n means that P_{n-1} triggers the attack. The state of the program is S_{n-1}. The following constraint must hold:

$$\gamma_M(S_{n-1}, P_{n-1}) \tag{4}$$

The SMT solver will return a model for packets $P_1, \ldots P_n$ such that the program after these receiving these packets reaches the attack point. Therefore, we run this procedure for varying lengths n to find network signatures of different lengths.

Also, this procedure only returns a particular signature of length n. We can generalize it by analyzing the transitions of M to generalize the signature. For example, if the i^{th} packet only matches certain predicates in the transition, we can generalize the packet on these bits (the rest of the bits are don't care). To generate the different signatures of length n, we can add blocking clauses for the signature we obtained to find a signature different from the current one.

6.3 Example

Let us consider a program receiving two-bit packets.

We denote the state variables by s_1, s_2 and packet variables by p_1, p_2. For the first `if` block, $\gamma : \neg s_1 \wedge \neg p_1$, and $\delta : s_1' \wedge s_2' \Leftrightarrow s_2$ (s_1 is set by the action). Similarly, for the second block: $\gamma : s_1 \wedge \neg s_2 \wedge p_2$, and $\delta : s_1' \Leftrightarrow s_1 \wedge s_2' \Leftrightarrow \neg s_2 \wedge p_1$.

```
s1 = 0;
s2 = 0;
while(True) {
        packet = socket.receive();
        if (s1 == 0 and packet[0] == 0) {
        s1 = 1;
        }
        if (s1 == 1 and s2 == 0 and packet[1] == 1) {
        s2 = (not s2) and packet[1];
        }
        if (s1 == 1 and s2 == 1 and packet[0] == 1
                                  and packet[1] == 1) {
        attack()
        }
}
```

For this program, initial state $S_0 = (0, 0)$. The attack point predicate γ_M is as follows:

$$s_1 \wedge s_2 \wedge p_1 \wedge p_1 \tag{5}$$

Therefore, if we add the constraints as described in the earlier section, for $n = 3$, we would obtain a valid network signature.

7 Evaluation

We present some preliminary evaluation of signature extraction using a synthetic malware example. We use the Z3 [5] SMT solver for solving the constraints generated for extracting signatures. We consider a malware program which processes 128 bit packets (i.e., 128 packet boolean variables). The malware program contains n state variables which are initialized to false. For every packet satisfying a certain condition, one of the state variables is set to true, and the malware program launches an attack when all state variables are set to true. Therefore, the signature length is n. We show the time to solve constraints to extract the signature with varying signature lengths in Figure 1.

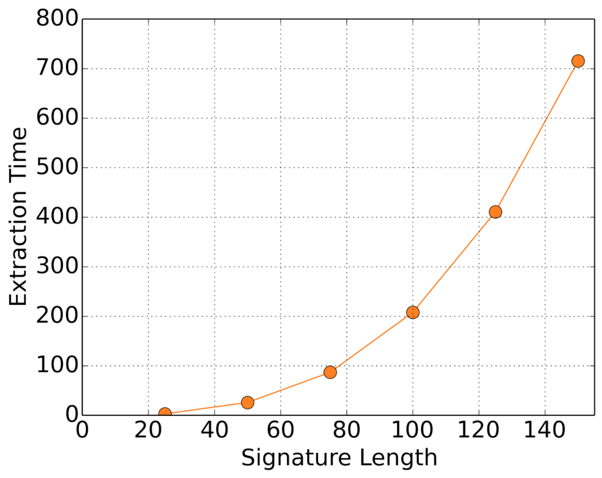

Fig. 1: Time taken to extract a signature using the Z3 solver with varying signature lengths.

8 Future Work

One of our simplifying assumptions is that we know the attack point in the malware. However, with large codebases, this may be difficult to identify manually. Therefore, an important extension is Identifying attack points automatically. Some initial approaches can be identifying statements that make accesses to restricted files or running restricted commands (e.g., sudo). To detect propagation in worms, we can find code which causes the malware to send itself on the network.

Another simplifying assumption is our malware program model as a set of if blocks in a while loop. We found real malware programs with large codebases with multiple files, thus packet processing can be across several procedures, requiring inter-procedural analyses. Also, it may be difficult to find malware code in the wild, only binaries, so extraction of signatures for a binary would be useful.

Another feature of malware code can be loops whose number of iterations depend on the bits of the packet. Thus, we cannot unroll the loop trivially, and require to find invariants for the loop predicated on packet characteristics.

Finally, the signature extraction can be used to verify implementations of receivers such that they adhere to a protocol specification.

References

[1] The case for network-based malware detection. `http://www.tmcnet.com/tmc/whitepapers/documents/whitepapers/2014/9599-case-network-based-malware-detection.pdf`.

[2] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 621–634, New York, NY, USA, 2009. ACM.

[3] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 317–329, New York, NY, USA, 2007. ACM.

[4] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex : protocol specification extraction. In *SP 2009, 30th IEEE Symposium on Security and Privacy, May 17-20, 2009, The Clare-*

mont Resort, Oakland, USA, Oakland, UNITED STATES, 05 2009.

[5] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] J. Oberheide, M. Bailey, and F. Jahanian. Polypack: An automated online packing service for optimal antivirus evasion. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, WOOT'09, pages 9–9, Berkeley, CA, USA, 2009. USENIX Association.

[7] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 3–3, Berkeley, CA, USA, 1998. USENIX Association.

[8] T. Teitelbaum. Codesurfer. *SIGSOFT Softw. Eng. Notes*, 25(1):99–, Jan. 2000.