Impeding Malware Analysis Using Conditional Code Obfuscation

Monirul Sharif[1], Andrea Lanzi[2],
Jonathon Giffin[1], Wenke Lee[1]

[1]Georgia Institute of Technology
[2]Universit`a degli Studi di Milano

NDSS 2008

Introduction

Rootkits Keyloggers Viruses

Bots Worms Spyware Trojans

Reverse engineering and Malware Analysis

Propagation Control Capabilities

Hundreds of new malware samples appear almost everyday…

Malware Malware Malware Malware Malware Malware

Automated analysis systems have become very important

Obfuscations that are easily applicable on existing code can be a threat

We present a **Simple**, **Automated** and **Transparent** Obfuscation against state-of-the-art malware analyzers

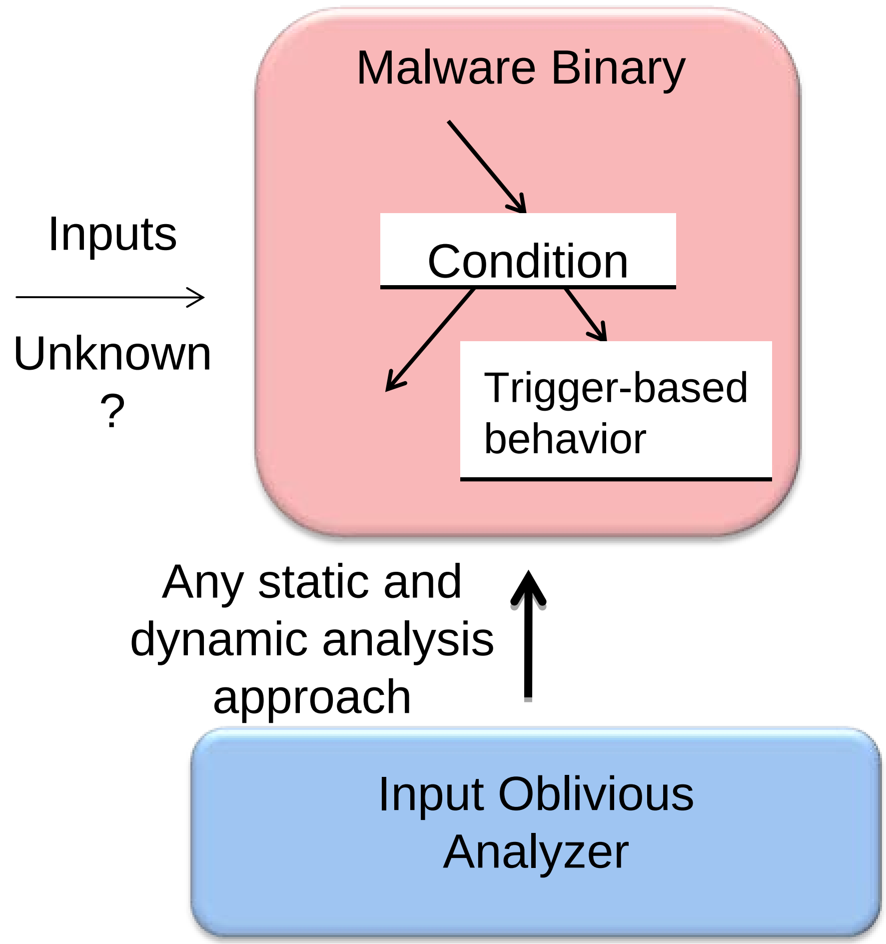
Malware Analysis and Obfuscations

Defense	**Offense**
Static Analysis based approaches	Polymorphism, metamorphism, packing, opaque predicates, anti-disassembly
← response	
Dynamic malware analysis	Trigger-based behavior (Logic bombs, time bombs, anti-debugging, anti-emulation, etc.)
← response	
Dynamic multipath exploration (Moser et al. 2007) Bitscope (Brumley et al. 2007) EXE (Cadar et al. 2006) Forced execution (Wilhelm et al. 2007)	? Conditional Code Obfuscation

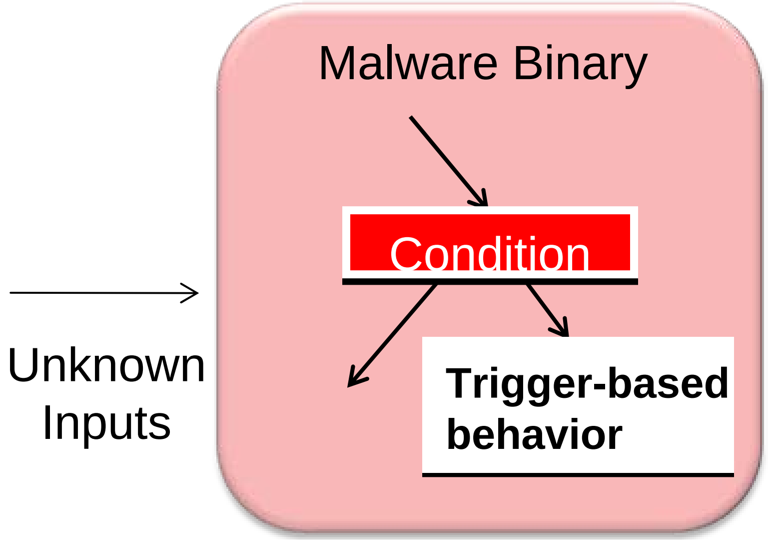
Rest of the Talk

- **Conditional Code Obfuscation**
  - Principles
  - Static analysis based automation
  - Automatic applicability on existing malware without modification
- **Implications**
  - Implications on Existing Analyzers
  - Measuring Obfuscation Strength
- **Prototype Implementation and Evaluation**
  - Evaluation on malware
- **Weaknesses and Defense**
  - How analysis can be improved to defender

Principles of Our Attack



Principles of Our Attack

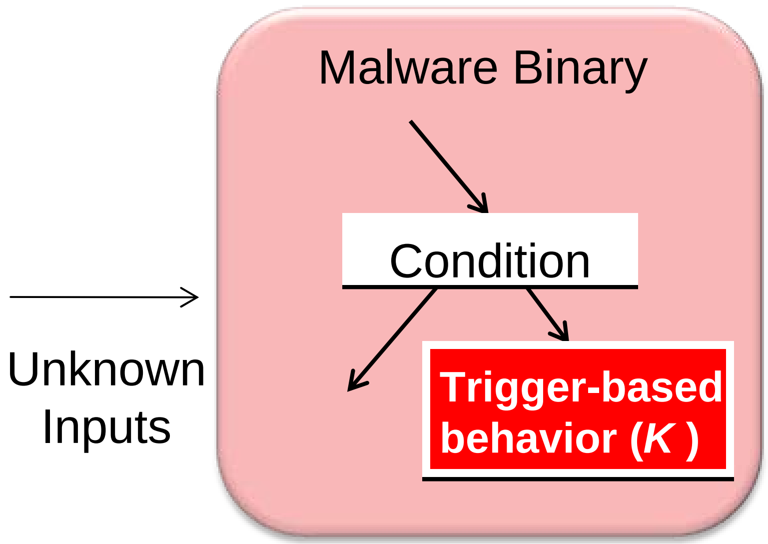



```
cmd = get_command(sock);
if (strcmp(cmd, "logkeys")==0))
{
      LogKeys()
}
```

↓

```
cmd = get_command(sock);
if (Hash(cmd)== H))
{
      LogKeys()
}
```

Principles of Our Attack

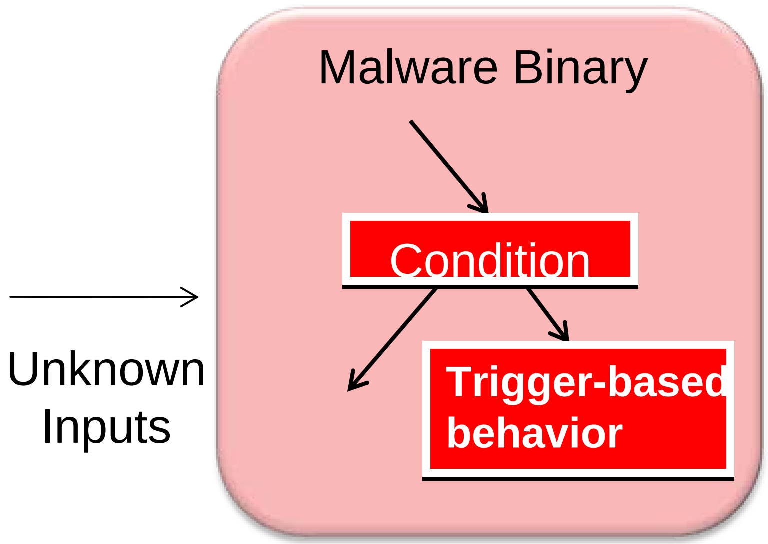



```
cmd = get_command(sock);
if (strcmp(cmd, “logkeys”)==0))
{
     LogKeys()
}
```

```
cmd = get_command(sock);
if (strcmp(cmd, “logkeys”)==0))
{
     decrypt(encr_LogKeys, K );
     encr_LogKeys()
}

encr_LogKeys(){
}
```

The key is inside the program

Principles of Our Attack

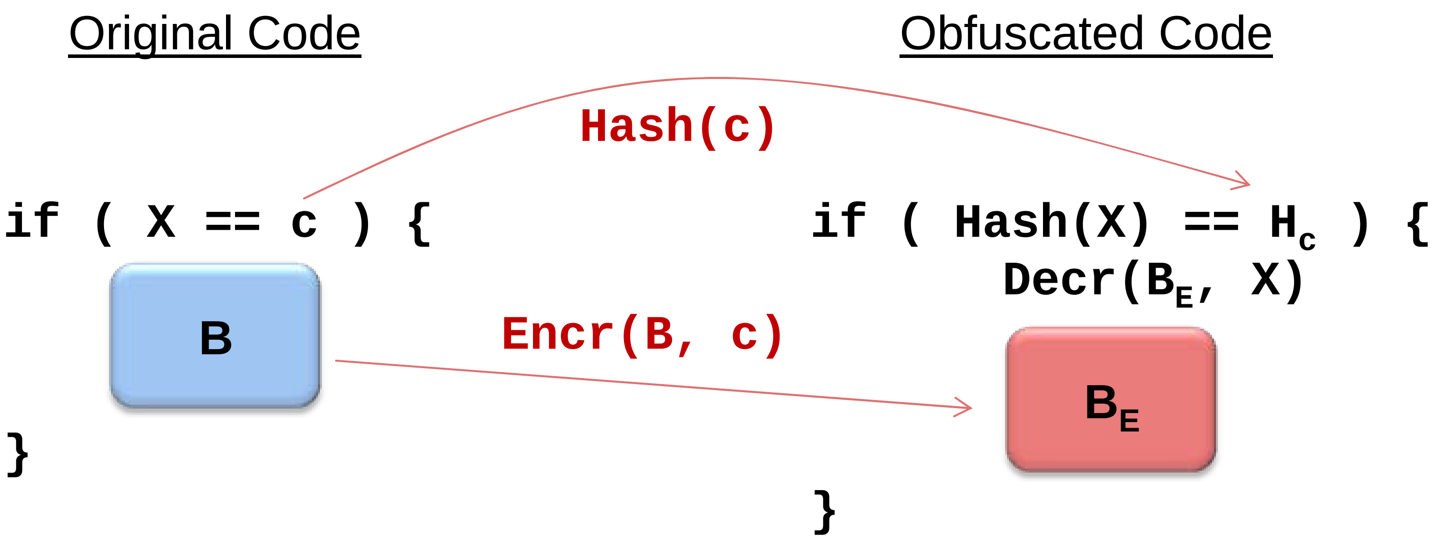



```
cmd = get_command(sock);
if (strcmp(cmd, "logkeys")==0))
{
     LogKeys()
}
```

```
cmd = get_command(sock);
if (Hash(cmd)== H))
{
     decrypt(encr_LogKeys, cmd);
     encr_LogKeys()
}

encr_LogKeys(){
}
```

The key is no longer inside the code

General Obfuscation Mechanism



- **Candidate Conditions - Conditions with equality**
  - The usual ‘==‘ operator
  - String equality checks – strcmp, memcmp, strncmp etc.
  - Conditions with ‘>’, ‘<‘, ‘!=‘ will not work
- **Conditional Code**
  - Any code that executes only when condition is satisfied

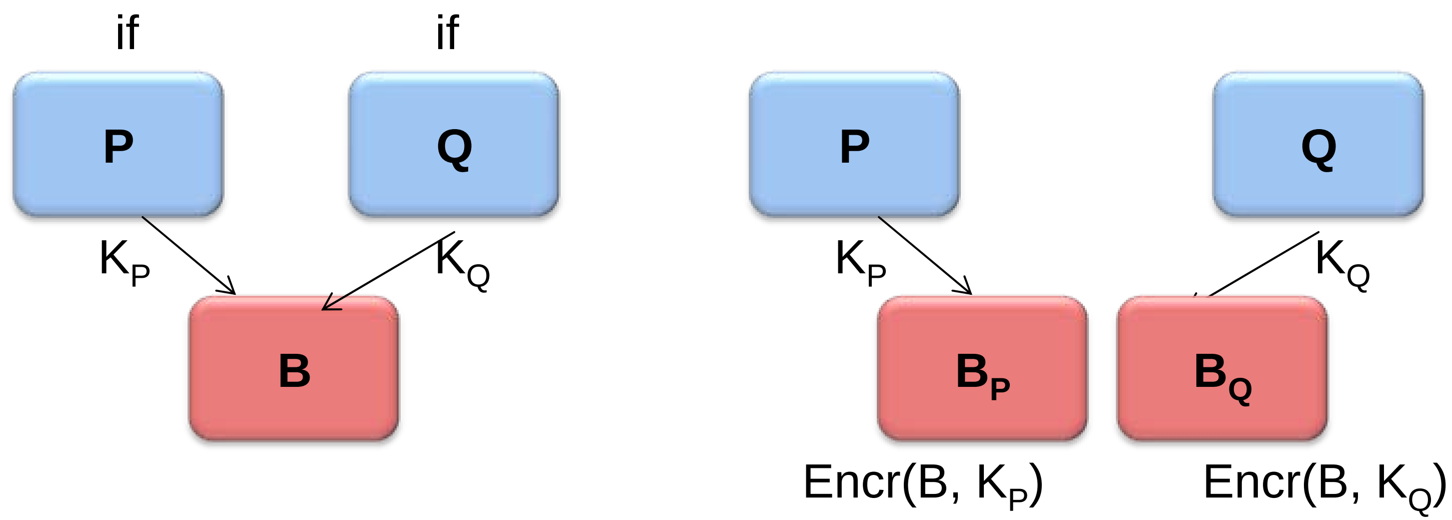
- Hash function Properties:
  - **Pre-image resistance** – Protects against reversing
    Hard to find c given H_c
  - **Second pre-image resistance** - Program correctness
    Hard to find another c’ where Hash(c’) = H_c

Automation Using Static Analysis

- **Identify Candidate Conditions**
  - Identify functions and create CFG for each function
  - Find blocks containing candidate conditions
- **Conditional code Identification**
  - Intra-procedural - Basic blocks **control dependen**t on condition with true outcome
  - Inter-procedural - Set of all functions ***only*** reachable from selected basic blocks
- **Exclude functions reachable from default path**
  - Conservative conditional code selection for function pointers

Automation Using Static Analysis

Handling Common Conditional Code



- Two keys are used in two paths. Duplicate code
- If one path is not candidate condition, no use in concealing the trigger code

Automation Using Static Analysis

Handling Complex Conditions

```
if ( X==a && Y==b ) {
      Attack()
}
```

```
if ( X==a ) {
  if (Y==b ) {
      Attack()
  }
}
```

Logical "and"

```
if ( X==a || Y==b ) {
      Attack()
}
```

```
if ( X==a )
  Attack()
}
else if (Y==b ) {
  Attack()
}
```

Logical "or"

Automation Using Static Analysis

Handling Complex Conditions

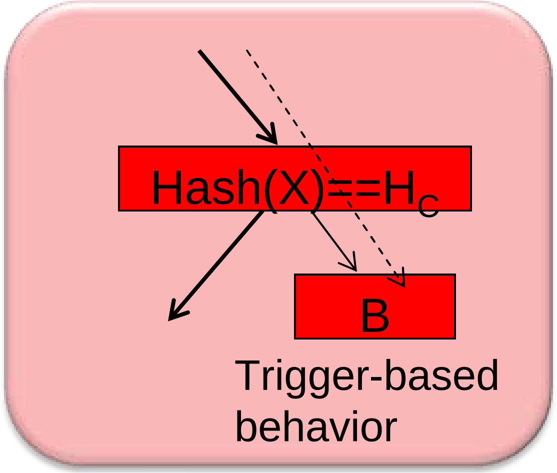
```
switch (cmd) {

  case 0:
    attack1();
    break;

  case 1:
    recon();

  case 2:
    attack2();

}
```

```
if (cmd==0)
  attack1();

if (cmd==1) {
  recon();
  attack2();
}

if (cmd==2)
  attack2();
```

Switch Case

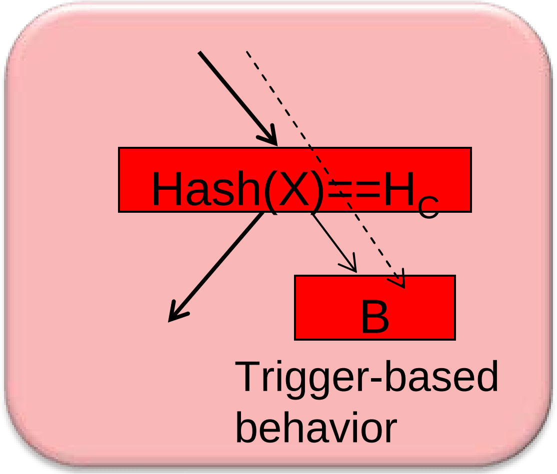
Consequences to Existing Analyzers

- Multi-Path Exploration (Moser et al., Bitscope)
  - Constraints are built for each path
  - Hash functions are non-linear, so cannot find solution
- Input Discovery (EXE)
  - Solves constraints to get inputs – symbolic execution
  - Same problem, cannot find derive input



Consequences to Existing Analyzers

- Forced Execution
  - Without solving constraints, forces execution
  - Without key, program crashes
- Static Analysis
  - Same as packed code, static analysis on trigger code is not possible

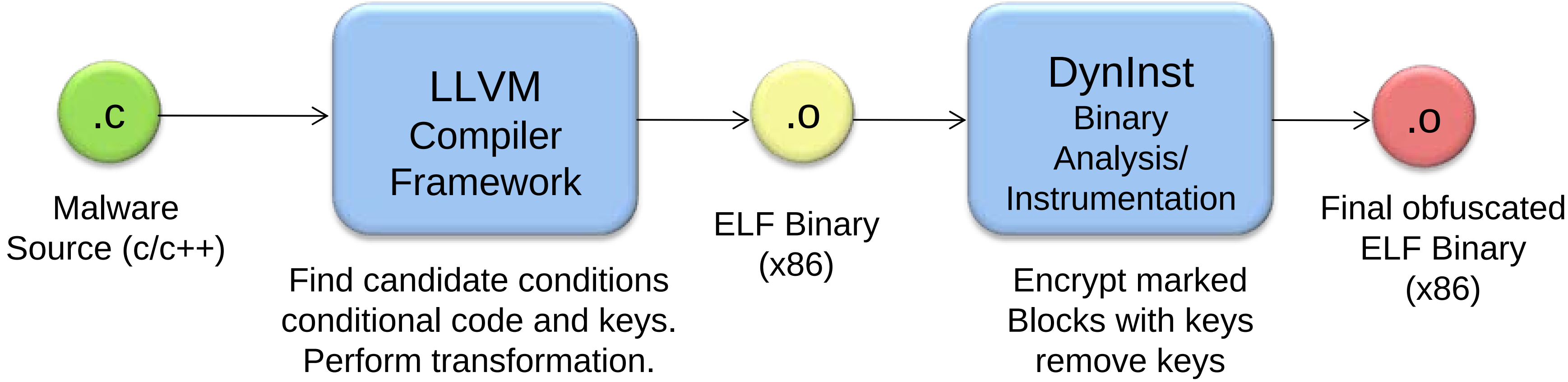


Attacks on the Obfuscation

- Attacks on $Hash(X)=H_c$
  - Find possible X for satisfying the above
- Input domain
  - Domain(X) – set of all possible values X may take
  - With time *t* for every hash computation,
    total time = Domain(X)*t*
  - For an integer I, Domain(I) = 2^{32}
- Brute Force attacks
- Dictionary Attacks

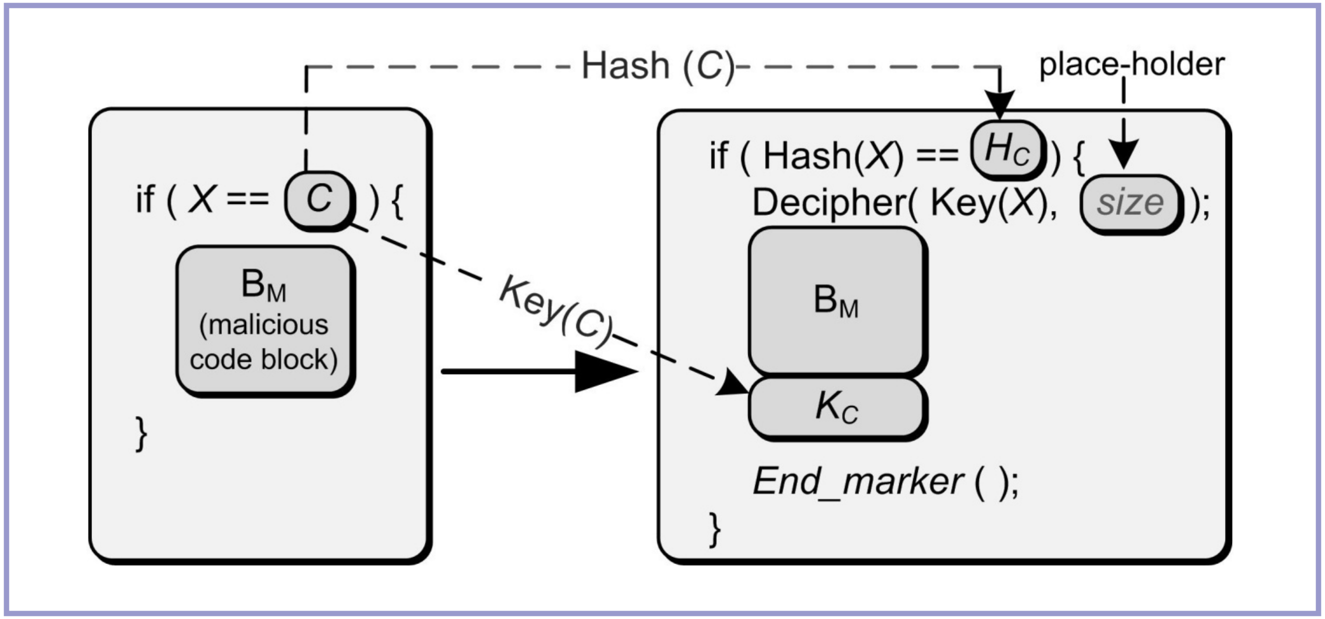
Prototype Implementation

- Overview
  - Implemented for Linux
  - Takes malware C source code and outputs obfuscated ELF binaries
- Analysis Level – both source code and binary levels required
  - Source and IR level – *type information is essential*
  - Binary level – *decrypted code must be executable*



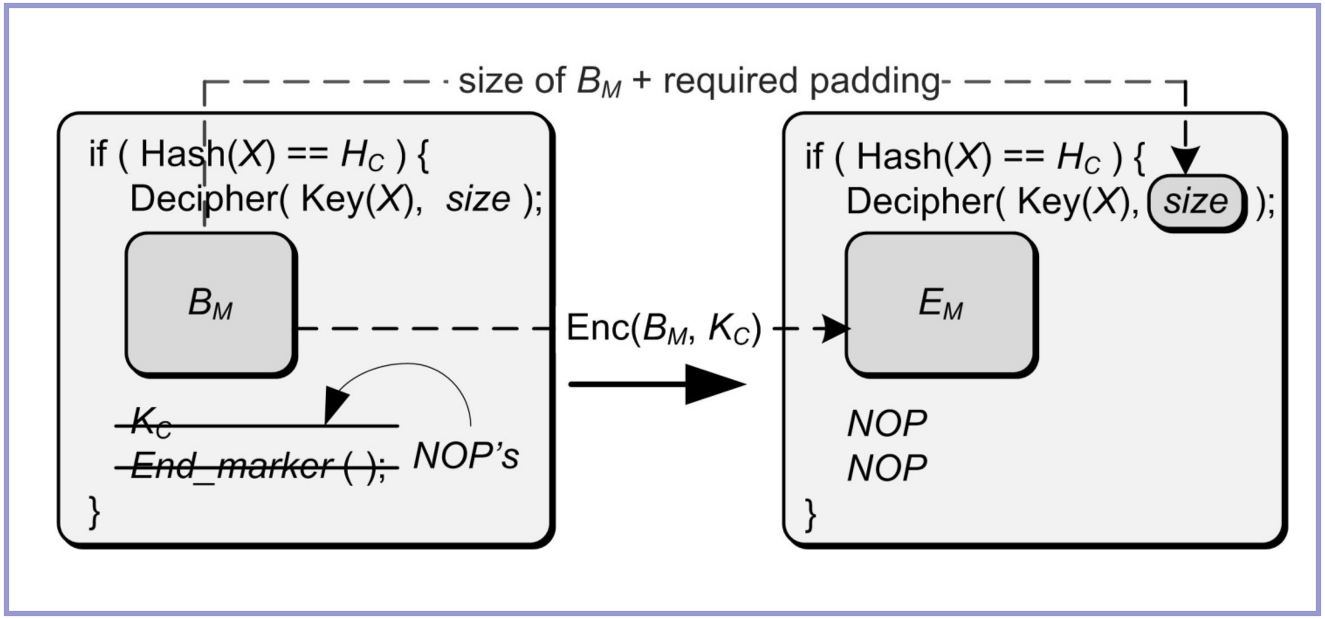
Simplified architectural view of the automated obfuscation system

Analysis and Transformation Phase



- Candidate Code Replacement
  - Enc(X)/Dec(X) Encryption/Decryption – AES with 256 bit keys
  - Hash function – Hash(X) - SHA-256
  - Different hash functions based on data type of X
- Decryption Keys and Markers
  - Key generation – Key(X) = Hash(X|N), N is Nonce

Encryption Phase



- DynInst based binary transformation tool
  - Finds Decipher(), and End_marker() and key (K_c)
  - Encrypts binary code with key
  - Removes marker and key from code

Experimental Results

- Evaluated by Obfuscating Malware Programs
  - Selected representative malware source programs for Linux with trigger based behavior

- Evaluation Method
  - Manually identified **malicious triggers** in malware
  - Applied obfuscation, counted how many were completely obfuscated by the automated system
  - Considered three levels of obfuscation strength –
    Strong – *strings*
    Medium – *integers*
    Weak – *booleans and return codes*

Experimental Results

Malware	Candidate Conditions	Malicious Triggers	Strong	Medium	Weak	None
Slapper Worm (P2P Engine)	157	28	-	28	-	-
Slapper Worm (Backdoor)	2	1	1	-	-	-
BotNET (IRC Botnet server)	61	52	52	-	-	-
passwd rootkit	5	2	2	-	-	-
login rootkit	19	3	2	-	-	1
top rootkit	17	2	-	-	-	2
chsh rootkit	10	4	2	-	2	-

Analysis of the Technique
(Strengths and Weaknesses)

- Knowledgable attacker can modify program to improve obfuscation effectiveness
  - Increase candidate conditions - replace <, >, != operators with ‘==‘
  - Increase conditional code – incorporate triggers that encapsulate more execution behavior
  - Increase input domains - Use variables with larger domains (e.g. strings) or use larger integers

- Weaknesses
  - Input domain may be very small in some cases
  - Upside on Malware detection – but polymorphic layers can be added

Defense Approaches

- **Incorporating cracking engine**
  - Equipped with decryptors where various keys are tried out repeatedly
  - Input domain knowledge (for dictionary attacks)
    - Determine type information – reduce domain space
    - Syscall return codes

- **Input-aware analysis**
  - Gather I/O traces along with malware binaries

Conclusion

- We presented an obfuscation technique that can be widely applicable on existing malware
- The obfuscation conceals trigger based behavior from existing and future analyzers
- We have shown its effectiveness on malware using our implemented automated prototype
- We presented its weaknesses and possible ways analyzers can be improved to defeat it

Thank you

Questions?

msharif@cc.gatech.edu

Experimental Results

Malware	Candidate Conditions	Original Size	Obufscated Size	% size increase
Slapper Worm (P2P Engine)	157	82.8 KB	97.3 KB	17%
Slapper Worm (Backdoor)	2	3.3 KB	10.7 KB	224%
BotNET (IRC Botnet server)	61	100.8 KB	115.1 KB	14%
passwd rootkit	5	6.9 KB	13.8 KB	172%
login rootkit	19	19.2 KB	27.3 KB	42%
top rootkit	17	43.9 KB	53.6 KB	22%
chsh rootkit	10	6.9 KB	14.3 KB	107%