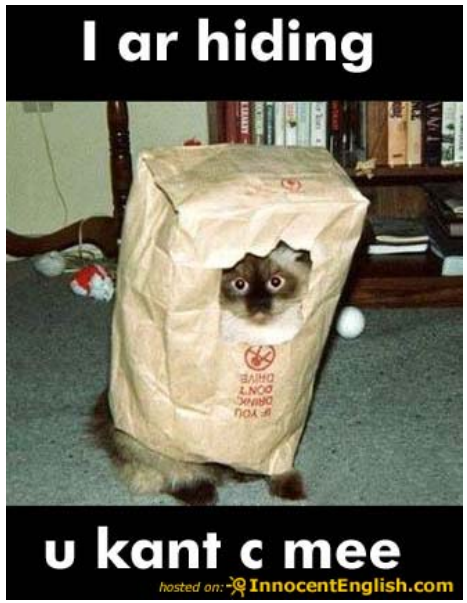
A Framework for Measuring Software Obfuscation Resilience Against Automated Attacks

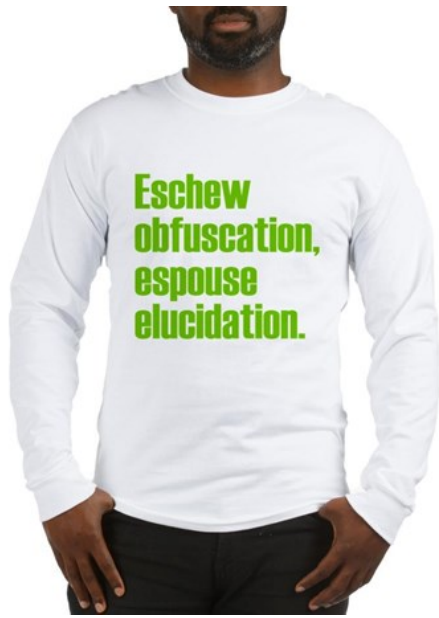
Sebastian Banescu, Martín Ochoa and Alexander Pretschner

Outline

Motivation

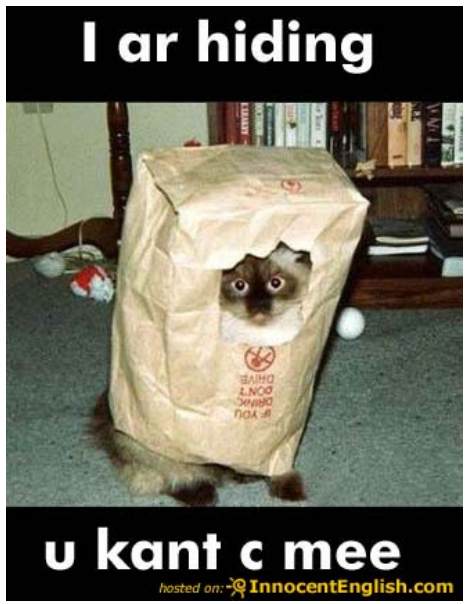
- Obfuscation used in practice both by good and bad guys
- Some call it security-by-obscurity
- Ideal: make obfuscation as strong as crypto, i.e. reduce security to a conjectured hard problem
- Has been done by indistinguishability obfuscation (unpractical)
- How about practical obfuscation transformations?
- Potency against manual attacks measured subjectively
- Not clear how to objectively compare effectiveness against automated attacks of different obfuscation transformations

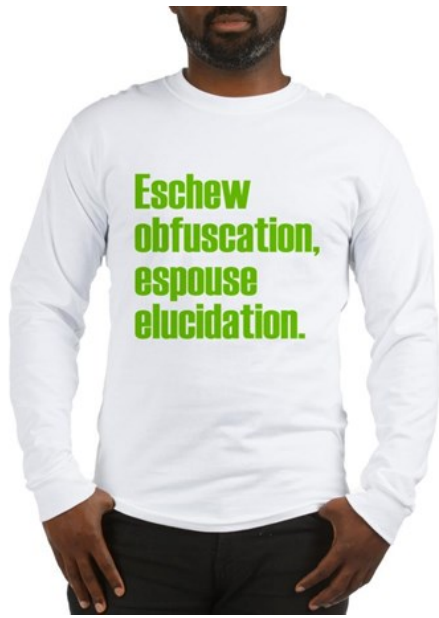




Motivation

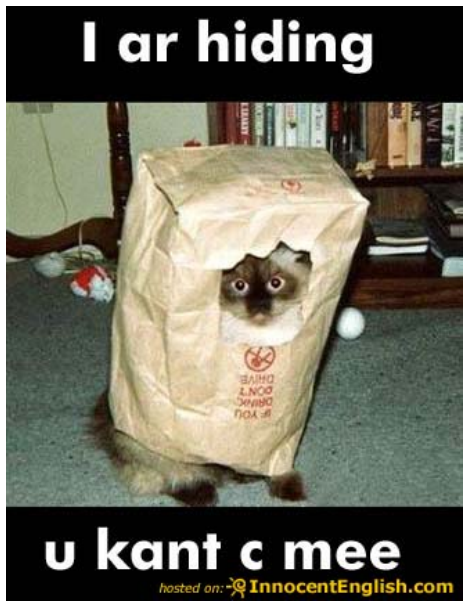
- Obfuscation used in practice both by good and bad guys
- Some call it security-by-obscurity
- Ideal: make obfuscation as strong as crypto, i.e. reduce security to a conjectured hard problem
- Has been done by indistinguishability obfuscation (unpractical)
- How about practical obfuscation transformations?
- Potency against manual attacks measured subjectively
- Not clear how to objectively compare effectiveness against automated attacks of different obfuscation transformations

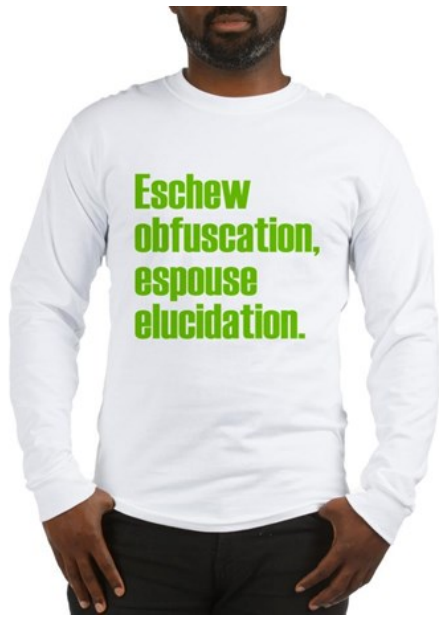




Motivation

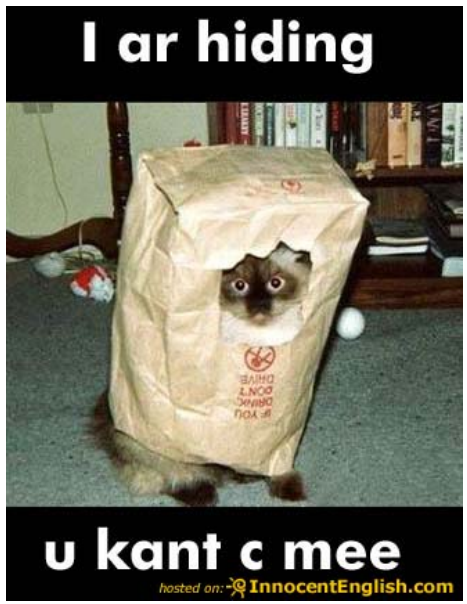
- Obfuscation used in practice both by good and bad guys
- Some call it security-by-obscurity
- Ideal: make obfuscation as strong as crypto, i.e. reduce security to a conjectured hard problem
- Has been done by indistinguishability obfuscation (unpractical)
- How about practical obfuscation transformations?
- Potency against manual attacks measured subjectively
- Not clear how to objectively compare effectiveness against automated attacks of different obfuscation transformations

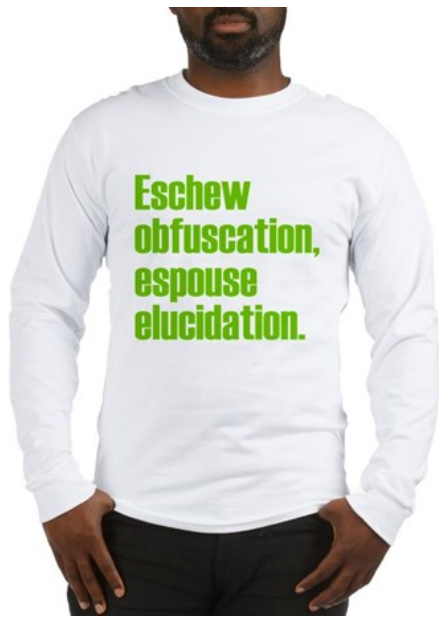




Motivation

- Obfuscation used in practice both by good and bad guys
- Some call it security-by-obscurity
- Ideal: make obfuscation as strong as crypto, i.e. reduce security to a conjectured hard problem
- Has been done by indistinguishability obfuscation (unpractical)
- How about practical obfuscation transformations?
- Potency against manual attacks measured subjectively
- Not clear how to objectively compare effectiveness against automated attacks of different obfuscation transformations





Goals of this work

- Objective measure of obfuscation resilience against automated attacks (de-obfuscation):
  - measure resilience of combinations of obfuscation transformations
  - measure resilience as a function of obfuscation transformation parameters
- Problem: The choice of automated attacks used is not objective
- Solution: Try multiple automated attacks and pick best results for each obfuscation transformation

Goals of this work

- Objective measure of obfuscation resilience against automated attacks (de-obfuscation):
  - measure resilience of combinations of obfuscation transformations
  - measure resilience as a function of obfuscation transformation parameters
- Problem: The choice of automated attacks used is not objective
- Solution: Try multiple automated attacks and pick best results for each obfuscation transformation

Two Different Perspectives

Defenders:

- Goals:
  - protect program control-flow (i.e. algorithms, intellectual property)
  - protect data embedded in program (e.g. hard-coded keys, passwords)
- Want lower-bound on attacker effort, increased via obfuscation transformations/parameters

Attackers:

- 2 classes of attacks (corresponding to each protection goal):
  - $\mathcal{A}_{CF}$ control-flow recovery attacks
  - $\mathcal{A}_{D}$ data recovery attacks
- Want to develop automated attacks outperforming prior known attacks (decrease upper bounds)

- Without fixing attackers by automation we cannot talk about bounds
- This work gives upper bounds for the lower bounds on effort of automated attacks against obfuscated programs
- We probe symbolic execution as an automated data recovery attack

Two Different Perspectives

Defenders:

- Goals:
  - protect program control-flow (i.e. algorithms, intellectual property)
  - protect data embedded in program (e.g. hard-coded keys, passwords)
- Want lower-bound on attacker effort, increased via obfuscation transformations/parameters

Attackers:

- 2 classes of attacks (corresponding to each protection goal):
  - $\mathcal{A}_{CF}$ control-flow recovery attacks
  - $\mathcal{A}_{D}$ data recovery attacks
- Want to develop automated attacks outperforming prior known attacks (decrease upper bounds)

- Without fixing attackers by automation we cannot talk about bounds
- This work gives upper bounds for the lower bounds on effort of automated attacks against obfuscated programs
- We probe symbolic execution as an automated data recovery attack

Two Different Perspectives

Defenders:

- Goals:
  - protect program control-flow (i.e. algorithms, intellectual property)
  - protect data embedded in program (e.g. hard-coded keys, passwords)
- Want lower-bound on attacker effort, increased via obfuscation transformations/parameters

Attackers:

- 2 classes of attacks (corresponding to each protection goal):
  - $\mathcal{A}_{CF}$ control-flow recovery attacks
  - $\mathcal{A}_{D}$ data recovery attacks
- Want to develop automated attacks outperforming prior known attacks (decrease upper bounds)

- Without fixing attackers by automation we cannot talk about bounds
- This work gives upper bounds for the lower bounds on effort of automated attacks against obfuscated programs
- We probe symbolic execution as an automated data recovery attack

Formal Model

- $\mathcal{P}$ universe of all executable programs
- $\mathcal{I}$, $\mathcal{O}$ program input, respectively output domains
- $\mathcal{T}$ universe of all obfuscation transformations applicable to $p \in \mathcal{P}$
- $[\![\cdot]\!]_{BB} : \mathcal{P} \to (\mathcal{I} \to \mathcal{O})$ *black-box* behavior of any program
- $\tau \in \mathcal{T}$ is a mapping $\tau : \mathcal{P} \to \mathcal{P}$ such that $[\![p]\!]_{BB} = [\![\tau(p)]\!]_{BB}$

Formal Model

- $\mathcal{P}$ universe of all executable programs
- $\mathcal{I}$, $\mathcal{O}$ program input, respectively output domains
- $\mathcal{T}$ universe of all obfuscation transformations applicable to $p \in \mathcal{P}$
- $[\![\cdot]\!]_{BB} : \mathcal{P} \rightarrow (\mathcal{I} \rightarrow \mathcal{O})$ *black-box* behavior of any program
- $\tau \in \mathcal{T}$ is a mapping $\tau : \mathcal{P} \rightarrow \mathcal{P}$ such that $[\![p]\!]_{BB} = [\![\tau(p)]\!]_{BB}$

Automated Data Recovery Attacks:

- $\mathcal{D}$ universe of data items from program binary or process memory
- $[\![\cdot]\!]_D : \mathcal{P} \rightarrow \mathcal{D}$ semantic characterization of data recovery
- $dif_D : \mathcal{D}^2 \rightarrow \mathbb{R}^+$ metric to compare similarity of 2 data items
- $T_D(s, \tau(p))$ shortest time needed by $\mathcal{A}_D$ having power s, to recover data item $d \in \mathcal{D}$ in program $p \in \mathcal{P}$, obfuscated with $\tau \in \mathcal{T}$

$$t[\mathcal{A}_D(\tau(p), s) = d \in \mathcal{D} \mid dif_D(d, [\![p]\!]_D) < \delta] \geq T_D(s, \tau(p))$$

Mapping Prior Works Onto Formal Model

- Several prior works presenting automated attacks on:
  - **virtualization obfuscation** ($\mathcal{T}_v \subset \mathcal{T}$): [Sharif et al., 2009, Guillot and Gazet, 2010, Coogan et al., 2011, Kinder, 2012]
  - **opaque predicates** ($\mathcal{T}_o \subset \mathcal{T}$): [Dalla Preda and Giacobazzi, 2005, Rolles, 2011]
  - **white-box cryptography** ($\mathcal{T}_w \subset \mathcal{T}$): [Billet et al., 2005, Wyseur et al., 2007, Michiels et al., 2009, Mulder et al., 2010]
  - **encoding literals** ($\mathcal{T}_{el} \subset \mathcal{T}$): [Guillot and Gazet, 2010, Gabriel, 2014]
  - **control-flow flattening** ($\mathcal{T}_{cff} \subset \mathcal{T}$): [Udupa et al., 2005]

- They fit into the formal model
- However, time needed to run automated attacks is missing because:
  - most works do not mention time needed for attacks in evaluation
  - no open-source implementation available to measure it ourselves

- Example of automated attack [Mulder et al., 2010] on white-box AES [Chow et al., 2003]:

$$t[\mathcal{A}_D(\tau_w(p), s) = d \in \mathcal{D} \mid dif_D(d, [\![p]\!]_D) = 0] \leq_{\mathcal{E}} 2^{22}/s,$$

Mapping Prior Works Onto Formal Model

- Several prior works presenting automated attacks on:
  - **virtualization obfuscation** ($\mathcal{T}_v \subset \mathcal{T}$): [Sharif et al., 2009, Guillot and Gazet, 2010, Coogan et al., 2011, Kinder, 2012]
  - **opaque predicates** ($\mathcal{T}_o \subset \mathcal{T}$): [Dalla Preda and Giacobazzi, 2005, Rolles, 2011]
  - **white-box cryptography** ($\mathcal{T}_w \subset \mathcal{T}$): [Billet et al., 2005, Wyseur et al., 2007, Michiels et al., 2009, Mulder et al., 2010]
  - **encoding literals** ($\mathcal{T}_{el} \subset \mathcal{T}$): [Guillot and Gazet, 2010, Gabriel, 2014]
  - **control-flow flattening** ($\mathcal{T}_{cff} \subset \mathcal{T}$): [Udupa et al., 2005]
- They fit into the formal model
- However, time needed to run automated attacks is missing because:
  - most works do not mention time needed for attacks in evaluation
  - no open-source implementation available to measure it ourselves
- Example of automated attack [Mulder et al., 2010] on white-box AES [Chow et al., 2003]:

$$t[\mathcal{A}_D(\tau_w(p), s) = d \in \mathcal{D} \mid dif_D(d, [\![p]\!]_D) = 0] \leq_{\mathcal{E}} 2^{22}/s,$$

Mapping Prior Works Onto Formal Model

- Several prior works presenting automated attacks on:
  - **virtualization obfuscation** ($\mathcal{T}_v \subset \mathcal{T}$): [Sharif et al., 2009, Guillot and Gazet, 2010, Coogan et al., 2011, Kinder, 2012]
  - **opaque predicates** ($\mathcal{T}_o \subset \mathcal{T}$): [Dalla Preda and Giacobazzi, 2005, Rolles, 2011]
  - **white-box cryptography** ($\mathcal{T}_w \subset \mathcal{T}$): [Billet et al., 2005, Wyseur et al., 2007, Michiels et al., 2009, Mulder et al., 2010]
  - **encoding literals** ($\mathcal{T}_{el} \subset \mathcal{T}$): [Guillot and Gazet, 2010, Gabriel, 2014]
  - **control-flow flattening** ($\mathcal{T}_{cff} \subset \mathcal{T}$): [Udupa et al., 2005]
- They fit into the formal model
- However, time needed to run automated attacks is missing because:
  - most works do not mention time needed for attacks in evaluation
  - no open-source implementation available to measure it ourselves
- Example of automated attack [Mulder et al., 2010] on white-box AES [Chow et al., 2003]:

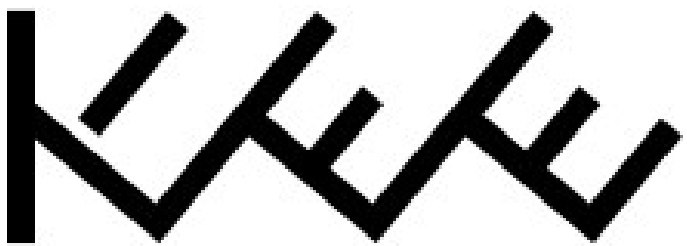
$$t[\mathcal{A}_D(\tau_w(p), s) = d \in \mathcal{D} \mid dif_D(d, [\![p]\!]_D) = 0] \leq_{\mathcal{E}} 2^{22}/s,$$

Case Study: Automated Data Retrieval with KLEE

- **Attacker Goal:** automated data recovery ($\mathcal{A}_D$)
- **Obfuscation transformations:** virtualization, opaque predicates, white-box cryptography, encoding literals
- **Obfuscation tool:** Tigress Diversifying C Virtualizer (v 1.3)
- **Automated attack tool:** KLEE symbolic execution engine
- **Disclaimer:** KLEE is not best attacker for all obfuscation transformations, but defenders should use it to measure resilience of their software against such an easy attack

Case Study: Automated Data Retrieval with KLEE

- **Attacker Goal:** automated data recovery ($\mathcal{A}_D$)
- **Obfuscation transformations:** virtualization, opaque predicates, white-box cryptography, encoding literals
- **Obfuscation tool:** Tigress Diversifying C Virtualizer (v 1.3)
- **Automated attack tool:** KLEE symbolic execution engine
- **Disclaimer:** KLEE is not best attacker for all obfuscation transformations, but defenders should use it to measure resilience of their software against such an easy attack

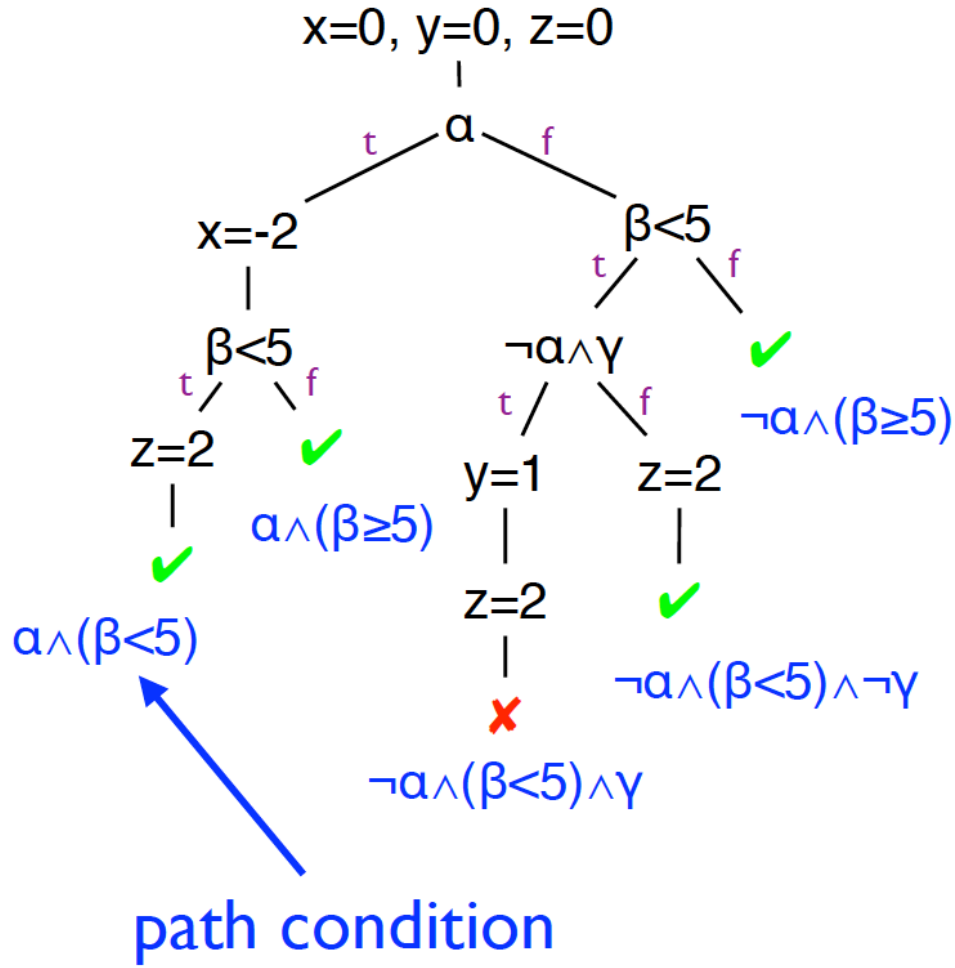


Case Study: Automated Data Retrieval with KLEE

- **Attacker Goal:** automated data recovery ($\mathcal{A}_D$)
- **Obfuscation transformations:** virtualization, opaque predicates, white-box cryptography, encoding literals
- **Obfuscation tool:** Tigress Diversifying C Virtualizer (v 1.3)
- **Automated attack tool:** KLEE symbolic execution engine
- **Disclaimer:** KLEE is not best attacker for all obfuscation transformations, but defenders should use it to measure resilience of their software against such an easy attack

Symbolic execution in one slide [Hicks, 2011]

```
1. int a = α, b = β, c = γ;
2.                    // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.   x = -2;
6. }
7. if (b < 5) {
8.   if (!a && c)  { y = 1; }
9.   z = 2;
10.}
11.assert(x+y+z!=3)
```



Simple License Checking Program

- First target program $p_1 \in \mathcal{P}$:

```
1 int main(int argc, char* argv[]) {
2   if (strcmp(argv[1], "my_license_key") == 0)
3     printf("The license key is correct!\n");
4   else
5     printf("The license key is incorrect!\n");
6   return 0;
7 }
```

Simple License Checking Program

- First target program $p_1 \in \mathcal{P}$:

```
1 int main(int argc, char* argv[]) {
2   if (strcmp(argv[1], "my_license_key") == 0)
3     printf("The license key is correct!\n");
4   else
5     printf("The license key is incorrect!\n");
6   return 0;
7 }
```

- $[\![p_1]\!]_D =$ "my_license_key" (string extraction via pattern matching)
- dif_D is string equality operator
- s power of attacker given by execution platform:
  - 2.8 GHz CPU, 4 GB memory
  - Ubuntu 14.04.1
  - LLVM 3.4.2

Simple License Checking Program

- First target program $p_1 \in \mathcal{P}$:

```
1 int main(int argc, char* argv[]) {
2   if (strcmp(argv[1], "my_license_key") == 0)
3     printf("The license key is correct!\n");
4   else
5     printf("The license key is incorrect!\n");
6   return 0;
7 }
```

- Problem 1: Directly applying virtualization obfuscation to p_1 vulnerable to string extraction via pattern matching, i.e. $[\![\tau_v(p_1)]\!]_D =$ "my_license_key"
- Solution 1: first apply literal encoding to eliminate hard-coded strings (see Figure on the right)

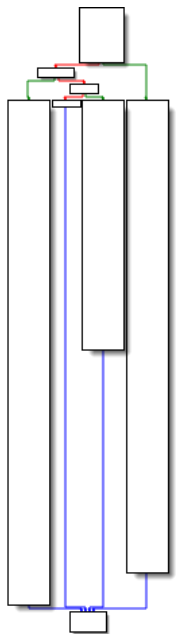


Figure : CFG of string encoding function in $\tau_{el}(p_1)$

Simple License Checking Program

- First target program $p_1 \in \mathcal{P}$:

```
1 int main(int argc, char* argv[]) {
2   if (strcmp(argv[1], "my_license_key") == 0)
3     printf("The license key is correct!\n");
4   else
5     printf("The license key is incorrect!\n");
6   return 0;
7 }
```

- Problem 2: $\tau_{el}(p_1)$ still easy to break via pre-processing and then string extraction via pattern matching
- Solution 2: apply virtualization to $\tau_{el}(p_1)$

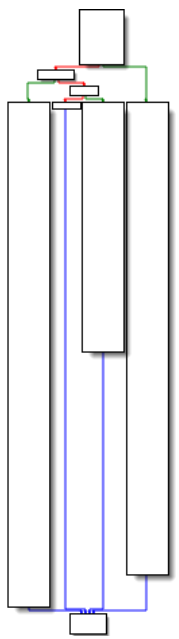


Figure : CFG of string encoding function in $\tau_{el}(p_1)$

Simple License Checking Program

- First target program $p_1 \in \mathcal{P}$:

```
1 int main(int argc, char* argv[]) {
2   if (strcmp(argv[1], "my_license_key") == 0)
3     printf("The license key is correct!\n");
4   else
5     printf("The license key is incorrect!\n");
6   return 0;
7 }
```

- Problem 2: $\tau_{el}(p_1)$ still easy to break via pre-processing and then string extraction via pattern matching
- Solution 2: apply virtualization to $\tau_{el}(p_1)$

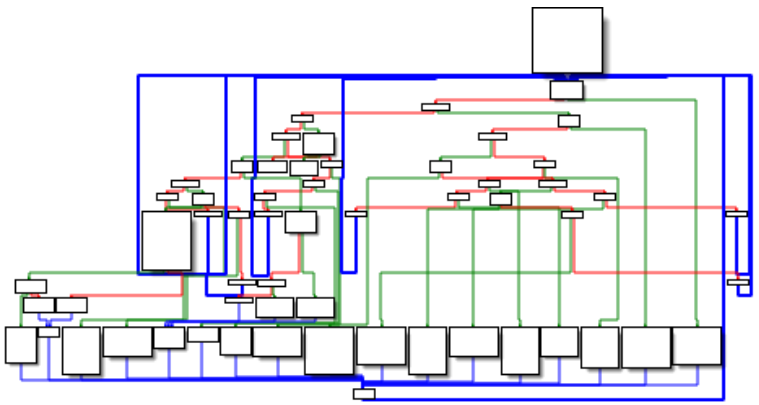


Figure : CFG of string encoding function in $\tau_v(\tau_{el}(p_1))$

Simple License Checking Program

- $\tau_v(\tau_{el}(p_1))$ has over 1300 LOC
- Attacker Goal: automatically extract key from $\tau_v(\tau_{el}(p_1))$
- Attacker assumption: license key could be of any length up to 32-bytes

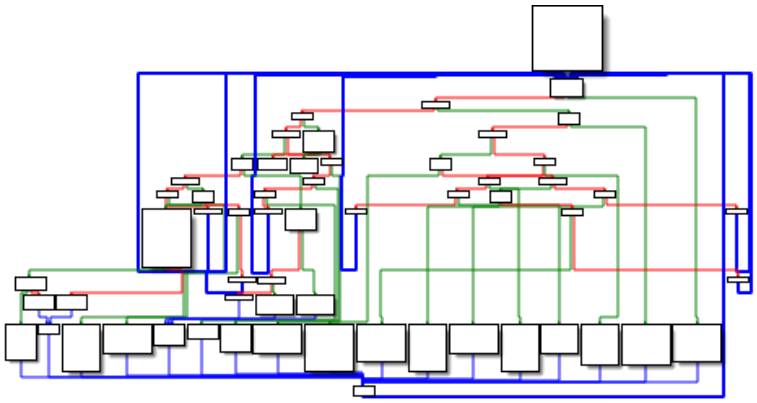


Figure : CFG of string encoding function in $\tau_v(\tau_{el}(p_1))$

- Attacker Steps:
  1. Run KLEE on $\tau_v(\tau_{el}(p_1))$ with a symbolic input of 32-bytes
  2. Find test case which causes $\tau_v(\tau_{el}(p_1))$ to output desired message
  3. The input used by that test case is the recovered data d s.t. $dif_D(d, [\![p_1]\!]_D) = 0$
     - Attack runtime put into formal model:

$$t[\mathcal{A}_D(\tau_v(\tau_{el}(p_1)), s) = d | dif_D(d, [\![p_1]\!]_D) = 0] \leq_{\mathcal{E}} 1.5sec$$

Simple License Checking Program

- $\tau_v(\tau_{el}(p_1))$ has over 1300 LOC
- Attacker Goal: automatically extract key from $\tau_v(\tau_{el}(p_1))$
- Attacker assumption: license key could be of any length up to 32-bytes

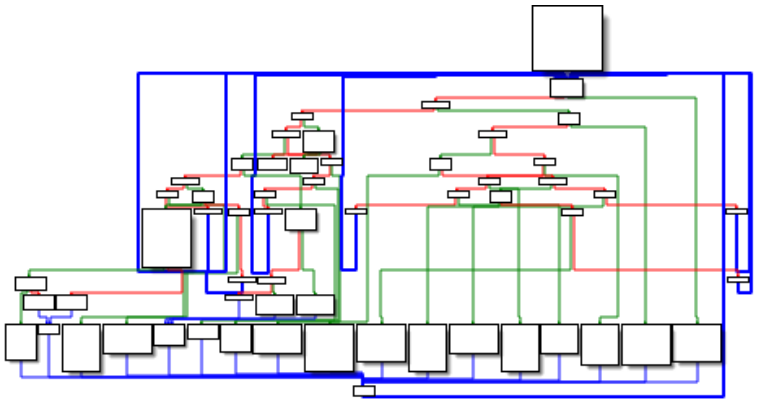


Figure : CFG of string encoding function in $\tau_v(\tau_{el}(p_1))$

- Attacker Steps:

1. Run KLEE on $\tau_v(\tau_{el}(p_1))$ with a symbolic input of 32-bytes
2. Find test case which causes $\tau_v(\tau_{el}(p_1))$ to output desired message
3. The input used by that test case is the recovered data d s.t. $dif_D(d, [\![p_1]\!]_D) = 0$

- Attack runtime put into formal model:

$$t[\mathcal{A}_D(\tau_v(\tau_{el}(p_1)), s) = d | dif_D(d, [\![p_1]\!]_D) = 0] \leq_{\mathcal{E}} 1.5sec$$

Simple License Checking Program

- $\tau_v(\tau_{el}(p_1))$ has over 1300 LOC
- Attacker Goal: automatically extract key from $\tau_v(\tau_{el}(p_1))$
- Attacker assumption: license key could be of any length up to 32-bytes

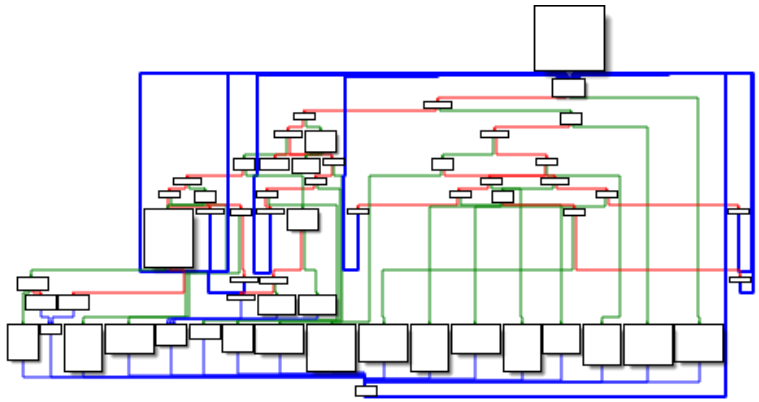


Figure : CFG of string encoding function in $\tau_v(\tau_{el}(p_1))$

- Attacker Steps:

1. Run KLEE on $\tau_v(\tau_{el}(p_1))$ with a symbolic input of 32-bytes
2. Find test case which causes $\tau_v(\tau_{el}(p_1))$ to output desired message
3. The input used by that test case is the recovered data d s.t. $dif_D(d, [\![p_1]\!]_D) = 0$

   - Attack runtime put into formal model:

$$t[\mathcal{A}_D(\tau_v(\tau_{el}(p_1)), s) = d | dif_D(d, [\![p_1]\!]_D) = 0] \leq_{\mathcal{E}} 1.5sec$$

Simple License Checking Program

- How is attack runtime affected by applying virtualization multiple times?:

$$t[\mathcal{A}_D(\tau_v(\tau_{el}(p_1)), s) = d | dif_D(d, [\![p_1]\!]_D) = 0] \leq_{\mathcal{E}} 1.5sec$$

$$t[\mathcal{A}_D(\tau_v^2(\tau_{el}(p_1)), s) = d | dif_D(d, [\![p_1]\!]_D) = 0] \leq_{\mathcal{E}} 8.8sec$$

$$t[\mathcal{A}_D(\tau_v^3(\tau_{el}(p_1)), s) = d | dif_D(d, [\![p_1]\!]_D) = 0] \leq_{\mathcal{E}} 780sec$$

- It has an exponential tendency given that:
  - $\text{LOC}(\tau_v(\tau_{el}(p_1)), s)) \approx 1300$
  - $\text{LOC}(\tau_v^2(\tau_{el}(p_1)), s)) \approx 3300$
  - $\text{LOC}(\tau_v^3(\tau_{el}(p_1)), s)) \approx 6600$

Simple License Checking Program

- How is attack runtime affected by adding opaque predicates to $\tau_v(\tau_{el}(p_1)), s)$?
- $\tau_o^{\alpha}(\tau_v(\,\cdot\,))$: adding α opaque predicates to each instruction handler of a virtualized program

$$t[\mathcal{A}_D(\tau_o^1(\tau_v(\tau_{el}(p_1))), s) = d | dif_D(d, [\![p_1]\!]_D) = 0] \leq_{\mathcal{E}} 1.5sec$$

$$t[\mathcal{A}_D(\tau_o^5(\tau_v(\tau_{el}(p_1))), s) = d | dif_D(d, [\![p_1]\!]_D) = 0] \leq_{\mathcal{E}} 1.6sec$$

$$t[\mathcal{A}_D(\tau_o^{10}(\tau_v(\tau_{el}(p_1))), s) = d | dif_D(d, [\![p_1]\!]_D) = 0] \leq_{\mathcal{E}} 1.7sec$$

$$t[\mathcal{A}_D(\tau_o^{20}(\tau_v(\tau_{el}(p_1))), s) = d | dif_D(d, [\![p_1]\!]_D) = 0] \leq_{\mathcal{E}} 2.3sec$$

- It has a logarithmic tendency given that:
  - $\text{LOC}(\tau_o^1(\tau_v(\tau_{el}(p_1)))) \approx 1300$
  - $\text{LOC}(\tau_o^5(\tau_v(\tau_{el}(p_1)))) \approx 1600$
  - $\text{LOC}(\tau_o^{10}(\tau_v(\tau_{el}(p_1)))) \approx 2100$
  - $\text{LOC}(\tau_o^{20}(\tau_v(\tau_{el}(p_1)))) \approx 7300$

More Complex License Checking Program

- Second target program $p_2 \in \mathcal{P}$:

```
 1 int main(int argc, char* argv[]) {
 2   unsigned long hash = 5381;
 3   unsigned char *str = argv[1];
 4
 5   while (int c = *str++)
 6     hash = ((hash << 5) + hash) + c;
 7
 8   if (((hash >> 32) == 0xbc150c6e) &&
 9       ((hash & 0xffffffff) == 0x49a54935))
10     printf("The license key is correct!\n");
11   else
12     printf("The license key is incorrect!\n");
13   return 0;
14 }
```

More Complex License Checking Program

- Second target program $p_2 \in \mathcal{P}$:

```
 1 int main(int argc, char* argv[]) {
 2   unsigned long hash = 5381;
 3   unsigned char *str = argv[1];
 4
 5   while (int c = *str++)
 6     hash = ((hash << 5) + hash) + c;
 7
 8   if (((hash >> 32) == 0xbc150c6e) &&
 9       ((hash & 0xffffffff) == 0x49a54935))
10     printf("The license key is correct!\n");
11   else
12     printf("The license key is incorrect!\n");
13   return 0;
14 }
```

- $[\![p_2]\!]_D =$ "my_license_key"
- dif_D is string equality operator
- s power of attacker given by execution platform:
  - 2.8 GHz CPU, 4 GB memory
  - Ubuntu 14.04.1
  - LLVM 3.4.2

More Complex License Checking Program

- Second target program $p_2 \in \mathcal{P}$:

```
 1 int main(int argc, char* argv[]) {
 2   unsigned long hash = 5381;
 3   unsigned char *str = argv[1];
 4
 5   while (int c = *str++)
 6     hash = ((hash << 5) + hash) + c;
 7
 8   if (((hash >> 32) == 0xbc150c6e) &&
 9       ((hash & 0xffffffff) == 0x49a54935))
10     printf("The license key is correct!\n");
11   else
12     printf("The license key is incorrect!\n");
13   return 0;
14 }
```

- Repeating previous attack steps, gives following runtimes:

$$t[\mathcal{A}_D(p_2, s) = d | dif_D(d, [\![p_2]\!]_D) = 0] \leq_{\mathcal{E}} 15min$$

$$t[\mathcal{A}_D(\tau_v(p_2), s) = d | dif_D(d, [\![p_2]\!]_D) = 0] \leq_{\mathcal{E}} 56min$$

- $LOC(\tau_v(p_2), s)) = 360$
- Note: found hash collisions `I)_NpMy1Aa!G` and `my_license_key`

Conclusions

- Proposed a framework for measuring resilience of obfuscation against
  - control-flow recovery attacks
  - data recovery attacks
- Discussed mapping prior works onto framework
- Instantiated model via case-study on data retrieval attacks
- Observations show that symbolic execution tools like KLEE:
  - are effective for data retrieval attacks from programs protected by literal encoding, virtualization and opaque predicates
  - have scalability issues when applying virtualization multiple times
  - can handle program which use non-linear hash functions instead of hard-coded secrets
  - are not effective for data retrieval attacks from programs protected by white-box cryptography

Conclusions

- Proposed a framework for measuring resilience of obfuscation against
  - control-flow recovery attacks
  - data recovery attacks
- Discussed mapping prior works onto framework
- Instantiated model via case-study on data retrieval attacks
- Observations show that symbolic execution tools like KLEE:
  - are effective for data retrieval attacks from programs protected by literal encoding, virtualization and opaque predicates
  - have scalability issues when applying virtualization multiple times
  - can handle program which use non-linear hash functions instead of hard-coded secrets
  - are not effective for data retrieval attacks from programs protected by white-box cryptography

Future Work

- Develop or use existing tools to perform systematic study of obfuscation resilience
- Measure runtimes of automated attacks as a function of multiple obfuscation transformations and their parameters
- Put shortest runtimes for each obfuscation transformation into mapping with following dimensions:
  - obfuscation transformation(s)
  - parameter values
  - automated attack technique/tool
  - program characteristics
- This would help practitioners pick obfuscation transformations for their programs/purposes

Future Work

- Develop or use existing tools to perform systematic study of obfuscation resilience
- Measure runtimes of automated attacks as a function of multiple obfuscation transformations and their parameters
- Put shortest runtimes for each obfuscation transformation into mapping with following dimensions:
  - obfuscation transformation(s)
  - parameter values
  - automated attack technique/tool
  - program characteristics
- This would help practitioners pick obfuscation transformations for their programs/purposes

Thank you for your attention

Questions ?


Billet, O., Gilbert, H., and Ech-Chatbi, C. (2005).
Cryptanalysis of a white box AES implementation.
In *Selected Areas in Cryptography*, number 3357 in LNCS, pages 227–240. Springer Berlin Heidelberg.


Chow, S., Eisen, P., Johnson, H., and Oorschot, P. C. V. (2003).
White-box cryptography and an AES implementation.
In *Selected Areas in Cryptography*, number 2595 in LNCS, pages 250–270. Springer Berlin Heidelberg.


Coogan, K., Lu, G., and Debray, S. (2011).
Deobfuscation of virtualization-obfuscated software: A semantics-based approach.
In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 275–284, New York, NY, USA. ACM.


Dalla Preda, M. and Giacobazzi, R. (2005).
Control code obfuscation by abstract interpretation.
In *Third IEEE International Conference on Software Engineering and Formal Methods.*, pages 301–310. IEEE.


Gabriel, F. (2014).
Deobfuscation: recovering an OLLVM-protected program.
http://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html.

Quarkslab, Accessed: 2014-01-20.


Guillot, Y. and Gazet, A. (2010).
Automatic binary deobfuscation.
Journal in computer virology, 6(3):261–276.


Kinder, J. (2012).
Towards static analysis of virtualization-obfuscated binaries.
In *19th Working Conference on Reverse Engineering (WCRE)*, pages 61–70.


Michiels, W., Gorissen, P., and Hollmann, H. D. L. (2009).
Cryptanalysis of a generic class of white-box implementations.
In Avanzi, R. M., Keliher, L., and Sica, F., editors, *Selected Areas in Cryptography*, number 5381 in LNCS, pages 414–428. Springer Berlin Heidelberg.


Mulder, Y. D., Wyseur, B., and Preneel, B. (2010).
Cryptanalysis of a perturbated white-box AES implementation.
In *Progress in Cryptology - INDOCRYPT 2010*, number 6498 in LNCS, pages 292–310. Springer Berlin Heidelberg.


Rolles, R. (2011).
Control Flow Deobfuscation via Abstract Interpretation.
`https://www.openrce.org/blog/view/1672/Control_Flow_Deobfuscation_via_Abstract_Interpretation`.
OpenRCE, Accessed: 2014-01-20.


Sharif, M., Lanzi, A., Giffin, J., and Lee, W. (2009).
Automatic reverse engineering of malware emulators.
In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 94–109.


Udupa, S., Debray, S., and Madou, M. (2005).
Deobfuscation: reverse engineering obfuscated code.
In *12th Working Conference on Reverse Engineering*.


Wyseur, B., Michiels, W., Gorissen, P., and Preneel, B. (2007).

Cryptanalysis of white-box DES implementations with arbitrary external encodings.
In *Selected Areas in Cryptography*, number 4876 in LNCS, pages 264–277. Springer Berlin Heidelberg.