Malware Makeover: Breaking ML-based Static Analysis by Modifying Executable Bytes

Keane Lucas, Mahmood Sharif, Lujo Bauer, Michael K. Reiter, Saurabh Shintre

Carnegie Mellon University

Duke

vmware

NortonLifeLock™

Malware detection is fundamental for cybersecurity

Anti-virus software routinely needs to examine programs for potential threats

https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/cloud-protection-microsoft-defender-antivirus?view=o365-worldwide
https://www.kaspersky.com/enterprise-security/wiki-section/products/machine-learning-in-cybersecurity
https://www.mcafee.com/enterprise/en-us/solutions/machine-learning.html
https://www.deepinstinct.com/

Malware detection is fundamental for cybersecurity

Anti-virus software routinely needs to examine programs for potential threats

Machine learning (ML) models show promise / are in use for detection

https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/cloud-protection-microsoft-defender-antivirus?view=o365-worldwide
https://www.kaspersky.com/enterprise-security/wiki-section/products/machine-learning-in-cybersecurity
https://www.mcafee.com/enterprise/en-us/solutions/machine-learning.html
https://www.deepinstinct.com/

Malware detection is fundamental for cybersecurity

Anti-virus software routinely needs to examine programs for potential threats

Machine learning (ML) models show promise / are in use for detection

But, malware classification models may be susceptible to evasion

https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/cloud-protection-microsoft-defender-antivirus?view=o365-worldwide
https://www.kaspersky.com/enterprise-security/wiki-section/products/machine-learning-in-cybersecurity
https://www.mcafee.com/enterprise/en-us/solutions/machine-learning.html
https://www.deepinstinct.com/

Malware detection is fundamental for cybersecurity

Anti-virus software routinely needs to examine programs for potential threats

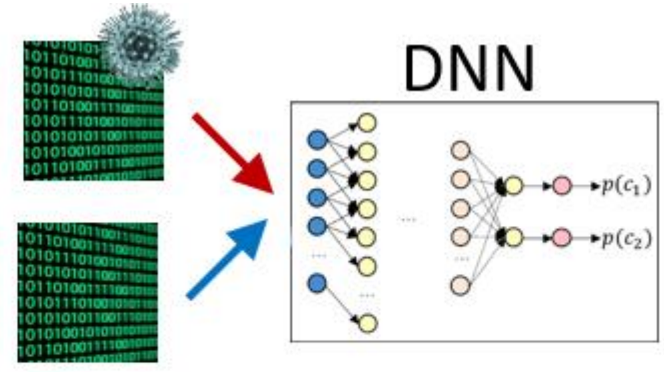
Machine learning (ML) models show promise / are in use for detection

But, malware classification models may be susceptible to evasion

Creating useful defenses requires knowledge of how ML models can be attacked

https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/cloud-protection-microsoft-defender-antivirus?view=o365-worldwide
https://www.kaspersky.com/enterprise-security/wiki-section/products/machine-learning-in-cybersecurity
https://www.mcafee.com/enterprise/en-us/solutions/machine-learning.html
https://www.deepinstinct.com/

Deep Neural Networks (DNNs) for Static Malware Detection

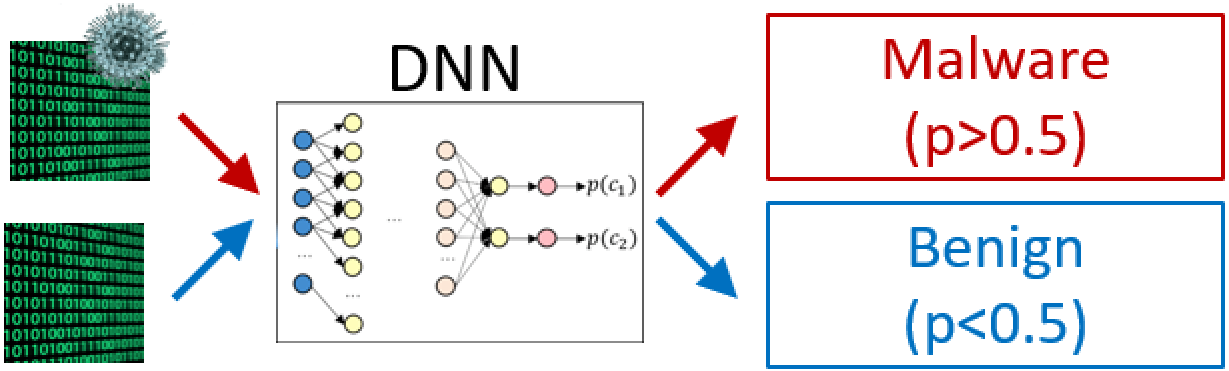


Program binary represented as variable length sequence of integers/bytes

- A single byte's meaning depends on the values of bytes around it
- Byte values are treated as categorical
    - Absolute difference between byte values has no meaning

E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas. 2017. "Malware Detection by Eating a Whole EXE." *arXiv [stat.ML]*. arXiv. http://arxiv.org/abs/1710.09435.

Deep Neural Networks (DNNs) for Static Malware Detection

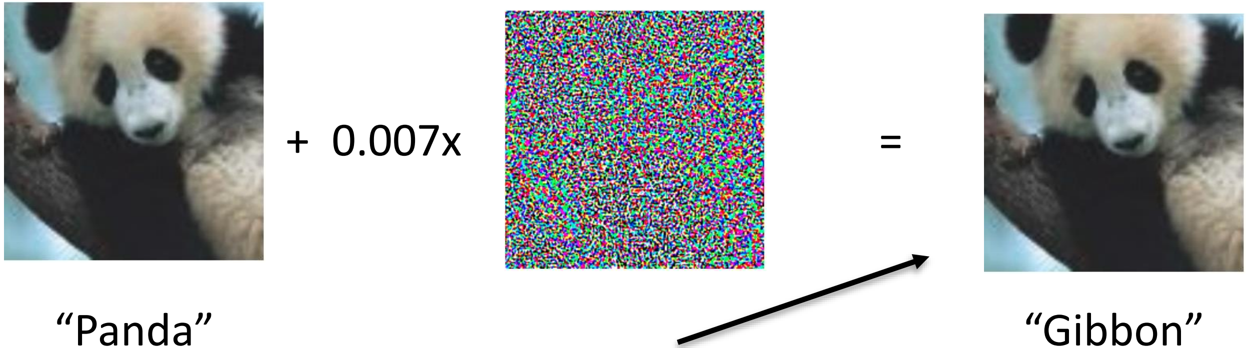


Program binary represented as variable length sequence of integers/bytes

- A single byte's meaning depends on the values of bytes around it
- Byte values are treated as categorical
    - Absolute difference between byte values has no meaning

E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas. 2017. "Malware Detection by Eating a Whole EXE." *arXiv [stat.ML]*. arXiv. http://arxiv.org/abs/1710.09435.

Attacking ML Algorithms – Adversarial Examples



"Panda"

"Gibbon"

Adversarial Example

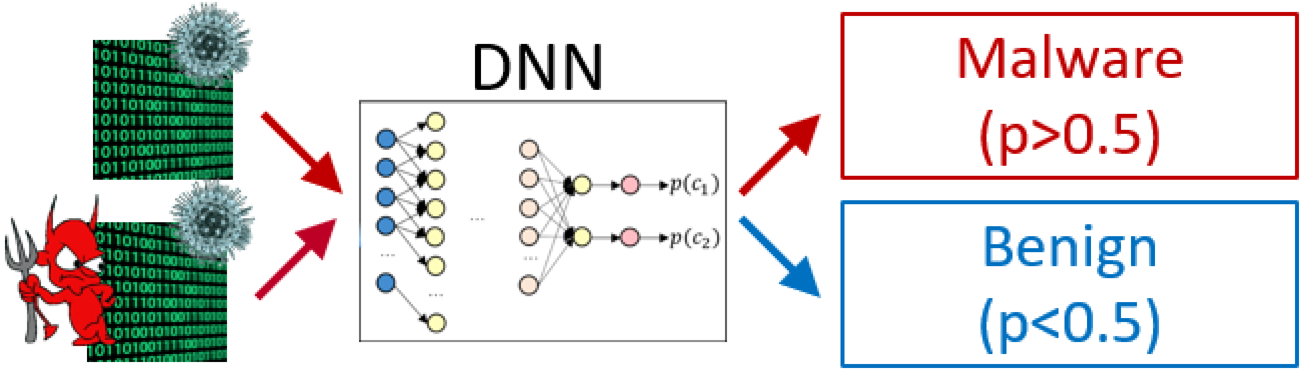
(image from Goodfellow 2015)

Attacks use classifier's trained weights to craft imperceptible adversarial noise (or perturbations) to cause misclassification

- Fast Gradient Sign Method (FGSM)
- Projected Gradient Descent (PGD)

I. J. Goodfellow, J. Shlens, and C. Szegedy. 2014. "Explaining and Harnessing Adversarial Examples." *arXiv [stat.ML]*. arXiv. http://arxiv.org/abs/1412.6572.

Attacking DNNs for Static Malware Detection



Must ensure all byte changes preserve binary functionality

Assume whitebox access to target model (can view trained weights)

- Our paper also examines a blackbox threat model

E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas. 2017. "Malware Detection by Eating a Whole EXE." *arXiv [stat.ML]*. arXiv. http://arxiv.org/abs/1710.09435.

Creating Adversarial Examples from Binaries

To modify binaries without changing functionality, use functionality preserving transformations:

V. Pappas, M. Polychronakis, and A. D. Keromytis. 2012. “Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization.” 2012. In Proc. IEEE S&P.
H. Koo and M. Polychronakis. 2016. “Juggling the gadgets: Binary-level code randomization using instruction displacement.” In Proc. AsiaCCS.

Creating Adversarial Examples from Binaries

To modify binaries without changing functionality, use functionality preserving transformations:

- In-Place Replacement (IPR)
  - Four types: preserv, swap, reorder, equiv

```
mov edx, [ebp+4]    (8b5504)
sub edx, -0x10      (83eaf0)
mov ebx, [ebp+8]    (8b5d08)
mov [ebx], edx      (8913)
```

```
mov ebx, [ebp+8]    (8b5d08)
mov edx, [ebp+4]    (8b5504)
sub edx, -0x10      (83eaf0)
mov [ebx], edx      (8913)
```

Reorder (1/4 IPR)

V. Pappas, M. Polychronakis, and A. D. Keromytis. 2012. "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization." 2012. In Proc. IEEE S&P.
H. Koo and M. Polychronakis. 2016. "Juggling the gadgets: Binary-level code randomization using instruction displacement." In Proc. AsiaCCS.

Creating Adversarial Examples from Binaries

To modify binaries without changing functionality, use functionality preserving transformations:

- In-Place-Replacement (IPR)
  - Four types: preserv, swap, reorder, equiv
- Displacement (Disp)

```
mov edx, [ebp+4]     (8b5504)
sub edx, -0x10       (83eaf0)
mov ebx, [ebp+8]     (8b5d08)
mov [ebx], edx       (8913)
```

```
mov ebx, [ebp+8]     (8b5d08)
mov edx, [ebp+4]     (8b5504)
sub edx, -0x10       (83eaf0)
mov [ebx], edx       (8913)
```

Reorder (1/4 IPR)

```
...        ...              ...
0x4587:    add ax, 0x10     (6683c010)
0x458b:    sub bx, 0x10     (6683eb10)
0x458f:    cmp ax, bx       (6639d8)
...        ...              ...
```

```
...        ...              ...
0x4587:    jmp 0x4800       (e974020000)
0x458c:    mov cx, cx       (6689c9)
0x458f:    cmp ax, bx       (6639d8)
...        ...              ...

...        ...              ...
0x4800:    add ax, 0x10     (6683c010)
0x4804:    sub bx, 0x10     (6683eb10)
0x4808:    nop              (90)
0x4805:    pushfd           (9c)
0x4806:    push ebx         (53)
0x4807:    add ebx, 0x1a    (83c31a)
0x480a:    pop ebx          (5b)
0x480b:    popfd            (9d)
0x480d:    jmp 0x458c       (e97afdffff)
...        ...              ...
```

Displacement

V. Pappas, M. Polychronakis, and A. D. Keromytis. 2012. "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization." 2012. In Proc. IEEE S&P.
H. Koo and M. Polychronakis. 2016. "Juggling the gadgets: Binary-level code randomization using instruction displacement." In Proc. AsiaCCS.

Attack Algorithm

1. Random initialization

Algorithm 1: White-box attack.

Input : $\mathbb{F} = \mathbb{H}(\mathbb{E}(\cdot))$, $\mathbb{L}_{\mathbb{F}}$, x, y, *niters*
Output : $\hat{x}$

$i \leftarrow 0$;
$\hat{x} \leftarrow RandomizeAll(x)$;

Attack Algorithm

1. Random initialization

2. For every function:

   a. Randomly choose from valid transformations

Algorithm 1: White-box attack.

Input : $\mathbb{F} = \mathbb{H}(\mathbb{E}(\cdot))$, $\mathbb{L}_{\mathbb{F}}$, x, y, $niters$
Output : $\hat{x}$

```
i ← 0;
x̂ ← RandomizeAll(x);
while F(x̂) = y and i < niters do
    for f ∈ x̂ do
        ê ← E(x̂);
        g ← ∂L_F(x̂, y) / ∂ê;
        o ← RandomTransformationType();
```

$i \leftarrow 0;$
$\hat{x} \leftarrow RandomizeAll(x);$
while $\mathbb{F}(\hat{x}) = y$ *and* $i < niters$ **do**
  for $f \in \hat{x}$ **do**
    $\hat{e} \leftarrow \mathbb{E}(\hat{x});$
    $g \leftarrow \frac{\partial \mathbb{L}_{\mathbb{F}}(\hat{x}, y)}{\partial \hat{e}};$
    $o \leftarrow RandomTransformationType();$

Attack Algorithm

1. Random initialization

2. For every function:

   a. Randomly choose from valid transformations
   b. Generate byte changes using chosen transformation and check gradient in embedding

Algorithm 1: White-box attack.

Input : $\mathbb{F} = \mathbb{H}(\mathbb{E}(\cdot))$, $\mathbb{L}_{\mathbb{F}}$, x, y, $niters$
Output : $\hat{x}$

$i \leftarrow 0$;
$\hat{x} \leftarrow RandomizeAll(x)$;
while $\mathbb{F}(\hat{x}) = y$ *and* $i < niters$ **do**
  for $f \in \hat{x}$ **do**
    $\hat{e} \leftarrow \mathbb{E}(\hat{x})$;
    $g \leftarrow \frac{\partial \mathbb{L}_{\mathbb{F}}(\hat{x}, y)}{\partial \hat{e}}$;
    $o \leftarrow RandomTransformationType()$;
    $\tilde{x} \leftarrow RandomizeFunction(\hat{x}, f, o)$;
    $\tilde{e} \leftarrow \mathbb{E}(\tilde{x})$;
    $\delta_f = \tilde{e}_f - \hat{e}_f$;

Guided Transformations

1. Random initialization

2. For every function:

    a. Randomly choose from valid transformations
    b. Generate byte changes using chosen transformation
    c. If byte changes align with loss gradient – accept and move on to next part of function. If not, discard and go back to step b
    d. Execute until all instructions in function have been reached

Algorithm 1: White-box attack.

Input : $\mathbb{F} = \mathbb{H}(\mathbb{E}(\cdot))$, $\mathbb{L}_{\mathbb{F}}$, x, y, $niters$
Output : $\hat{x}$

$i \leftarrow 0;$
$\hat{x} \leftarrow RandomizeAll(x);$
while $\mathbb{F}(\hat{x}) = y$ *and* $i < niters$ **do**
  for $f \in \hat{x}$ **do**
    $\hat{e} \leftarrow \mathbb{E}(\hat{x});$
    $g \leftarrow \frac{\partial \mathbb{L}_{\mathbb{F}}(\hat{x}, y)}{\partial \hat{e}};$
    $o \leftarrow RandomTransformationType();$
    $\tilde{x} \leftarrow RandomizeFunction(\hat{x}, f, o);$
    $\tilde{e} \leftarrow \mathbb{E}(\tilde{x});$
    $\delta_f = \tilde{e}_f - \hat{e}_f;$
    if $g_f \cdot \delta_f > 0$ **then**
      $\hat{x} \leftarrow \tilde{x};$
    end

Attack Algorithm

1. Random initialization
2. For every function:
   a. -- d. …
3. Repeat step 2 until success or 200 iterations

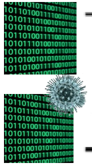
Algorithm 1: White-box attack.

Input : $\mathbb{F} = \mathbb{H}(\mathbb{E}(\cdot))$, $\mathbb{L}_{\mathbb{F}}$, x, y, $niters$
Output : $\hat{x}$

$i \leftarrow 0$;
$\hat{x} \leftarrow RandomizeAll(x)$;
while $\mathbb{F}(\hat{x}) = y$ *and* $i < niters$ **do**
  for $f \in \hat{x}$ **do**
    $\hat{e} \leftarrow \mathbb{E}(\hat{x})$;
    $g \leftarrow \frac{\partial \mathbb{L}_{\mathbb{F}}(\hat{x}, y)}{\partial \hat{e}}$;
    $o \leftarrow RandomTransformationType()$;
    $\tilde{x} \leftarrow RandomizeFunction(\hat{x}, f, o)$;
    $\tilde{e} \leftarrow \mathbb{E}(\tilde{x})$;
    $\delta_f = \tilde{e}_f - \hat{e}_f$;
    if $g_f \cdot \delta_f > 0$ **then**
      $\hat{x} \leftarrow \tilde{x}$;
    end
  end
  $i \leftarrow i + 1$;
end
return $\hat{x}$;

Experiment Setup – Dataset

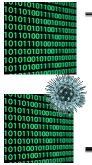
- 32-bit portable executable (PE) files, smaller than 5 MB, first seen in 2020, collected from VirusTotal feed (*VTFeed*), either 0 or >40 AV detections



VTFeed	Train	Val.	Test
Benign	111,258	13,961	13,926
Malicious	111,395	13,870	13,906

Experiment Setup – Dataset

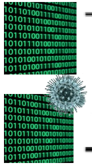
- 32-bit portable executable (PE) files, smaller than 5 MB, first seen in 2020, collected from VirusTotal feed (*VTFeed*), either 0 or >40 AV detections
- Labeled as benign (resp. malicious) if classified malicious by 0 (resp. >40) antivirus vendors aggregated by VirusTotal



VTFeed	Train	Val.	Test
Benign	111,258	13,961	13,926
Malicious	111,395	13,870	13,906

Experiment Setup – Dataset

- 32-bit portable executable (PE) files, smaller than 5 MB, first seen in 2020, collected from VirusTotal feed (*VTFeed*), either 0 or >40 AV detections
- Labeled as benign (resp. malicious) if classified malicious by 0 (resp. >40) antivirus vendors aggregated by VirusTotal
- 139K benign and 139K malicious, shuffled, and randomly partitioned into Train (80%), Validation (10%), and Test (10%) sets

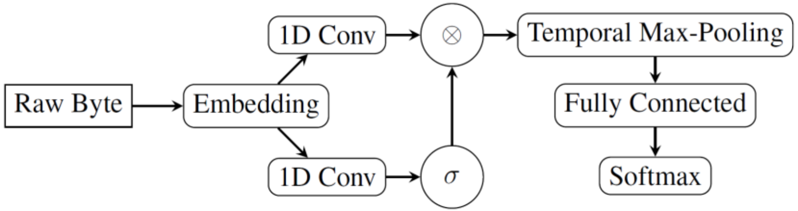


VTFeed	Train	Val.	Test
Benign	111,258	13,961	13,926
Malicious	111,395	13,870	13,906

Experiment Setup – DNNs

State-of-the-art architectures we trained:

- MalConv – proposed by Raff et al.
- Avast – proposed by Krčál et al.



Architecture diagram of MalConv model (from Raff et al.)

	Accuracy Train	Accuracy Val.	Accuracy Test	TPR @ 0.1% FPR
AvastNet	99.89%	98.59%	98.60%	94.78%
MalConv	99.97%	98.67%	98.53%	96.08%

Endgame – pre-trained DNN (Anderson et al.)

- Based on MalConv architecture
- Trained on 600K binaries, evenly distributed between benign and malicious
- 92% detection rate when restricted to a false positive rate of 0.1%

H. S. Anderson and P. Roth. 2018. Ember: An Open Dataset for Training Static PE Malware Machine Learning Models .arXiv preprint arXiv:1804.04637(2018).
M. Krcál et al. “Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only.” ICLR (2018).
E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas. 2017. “Malware Detection by Eating a Whole EXE.” *arXiv [stat.ML]*. arXiv. http://arxiv.org/abs/1710.09435.

Results – DNNs and Malware Samples

Malware samples used to construct adversarial examples

- 100 sampled from VirusTotal (aggregates binaries and anti-virus vendor detections)
    - Unpacked
    - Size below models' smallest input (512KB)
    - At least 40 anti-virus detections for malware

Experiment Setup – Measuring Success

Experiment methods

- 10 repetitions of each experiment
- Deemed successful if an attack can reduce maliciousness score to below 0.1% FPR threshold (0.5 for Endgame)

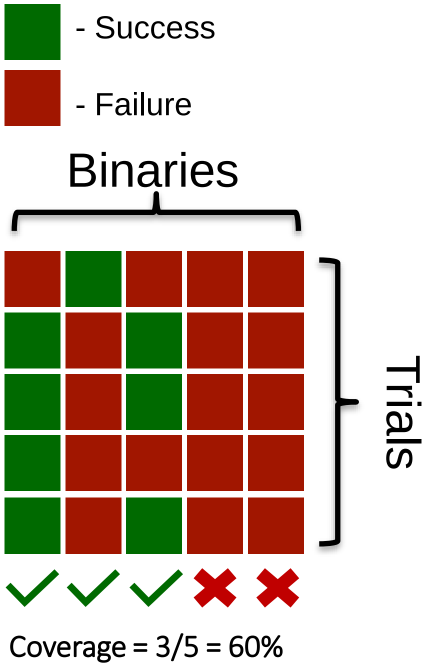
Experiment Setup – Measuring Success

Experiment methods

- 10 repetitions of each experiment
- Deemed successful if an attack can reduce maliciousness score to below 0.1% FPR threshold (0.5 for Endgame)

Two measures of success

- Coverage – fraction of *binaries* an attack was successful in *at least* one of the trials



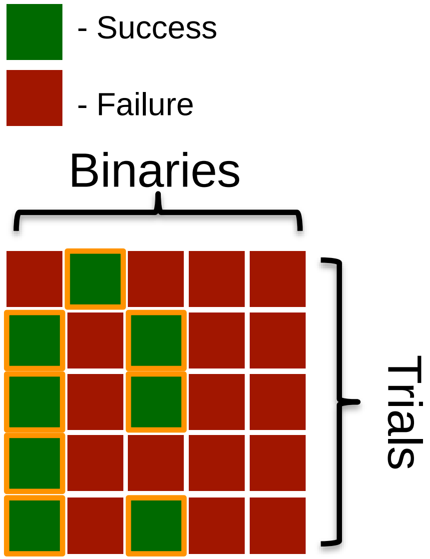
Experiment Setup – Measuring Success

Experiment methods

- 10 repetitions of each experiment
- Deemed successful if an attack can reduce maliciousness score to below 0.1% FPR threshold (0.5 for Endgame)

Two measures of success

- Coverage – fraction of *binaries* an attack was successful in *at least* one of the trials
- Potency – fraction of *trials* that succeeded, over all binaries



Coverage = 3/5 = 60%
Potency = 8/25 = 32%

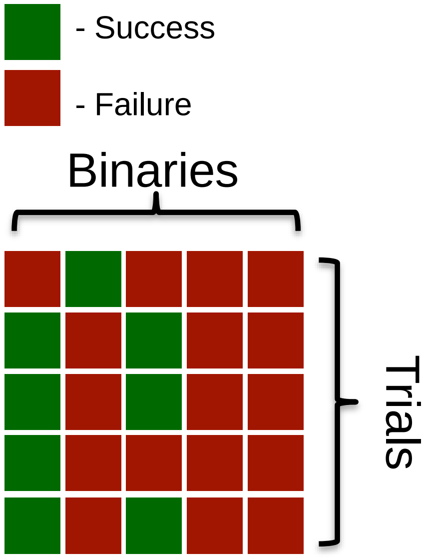
Experiment Setup – Measuring Success

Experiment methods

- 10 repetitions of each experiment
- Deemed successful if an attack can reduce maliciousness score to below 0.1% FPR threshold (0.5 for Endgame)

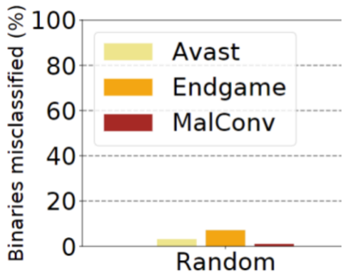
Two measures of success

- Coverage – fraction of *binaries* an attack was successful in *at least* one of the trials
- Potency – fraction of *trials* that succeeded, over all binaries



Coverage = 3/5 = 60%
Potency = 8/25 = 32%
Coverage $\geq$ Potency

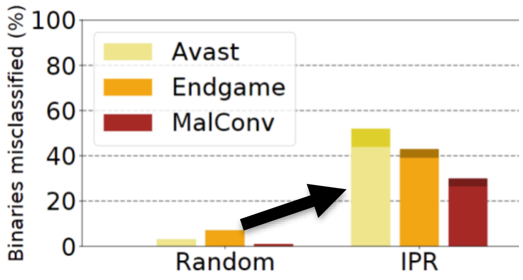
Results – Overall



Attack success rates in the white-box setting

- Potency shown as lighter bars and coverage as darker bars

Results – Overall

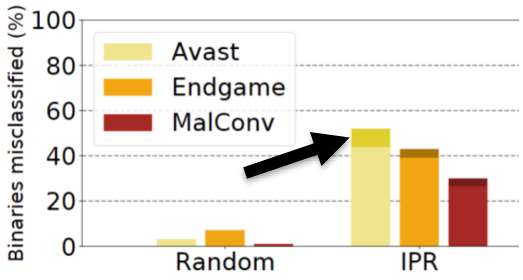


Attack success rates in the white-box setting

- Potency shown as lighter bars and coverage as darker bars

Random < IPR

Results – Overall

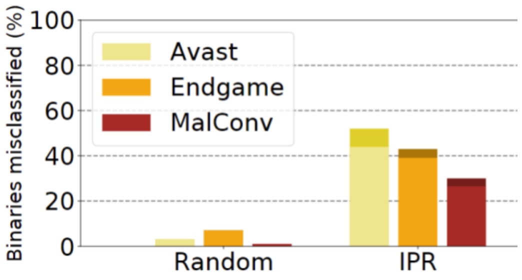


Attack success rates in the white-box setting

- Potency shown as lighter bars and coverage as darker bars

Random < IPR

Results – Overall

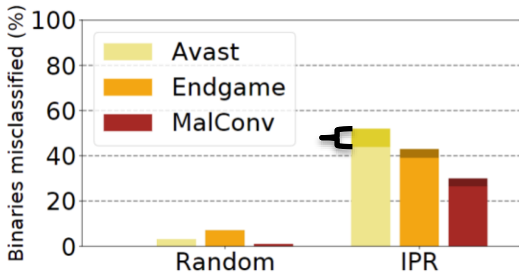


Attack success rates in the white-box setting

- Potency shown as lighter bars and coverage as darker bars

Random < IPR

Results – Overall

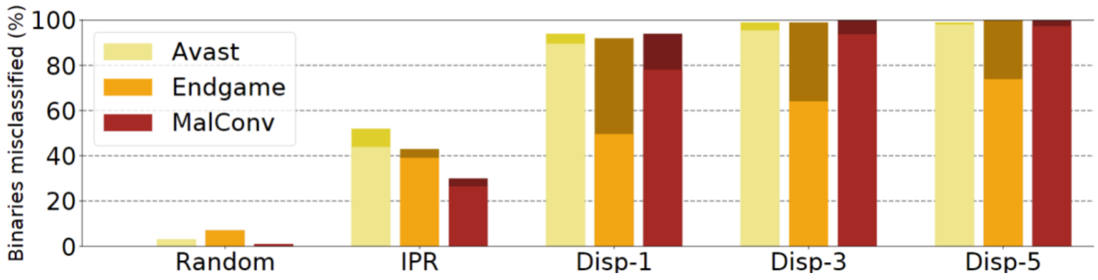


Attack success rates in the white-box setting

- Potency shown as lighter bars and coverage as darker bars

Random < IPR

Results – Overall

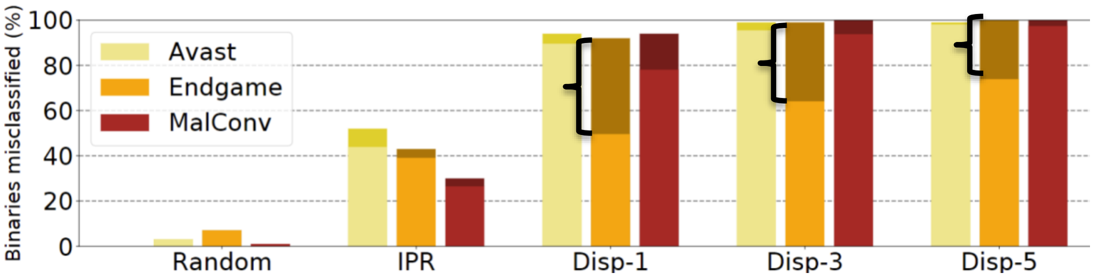


Attack success rates in the white-box setting

- Potency shown as lighter bars and coverage as darker bars

Random < IPR < Disp

Results – Overall

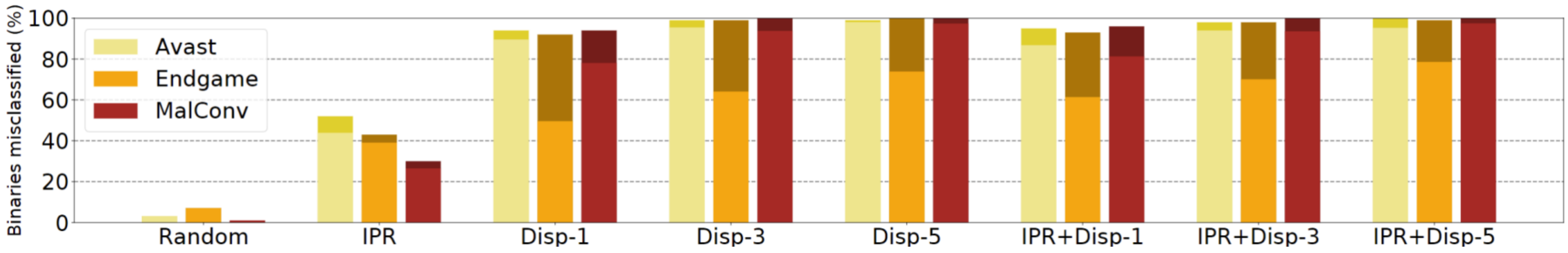


Attack success rates in the white-box setting

- Potency shown as lighter bars and coverage as darker bars

Random < IPR < Disp

Results – Overall



Attack success rates in the white-box setting

- Potency shown as lighter bars and coverage as darker bars

Random < IPR < Disp < IPR+Disp

Results – Attack Behavior

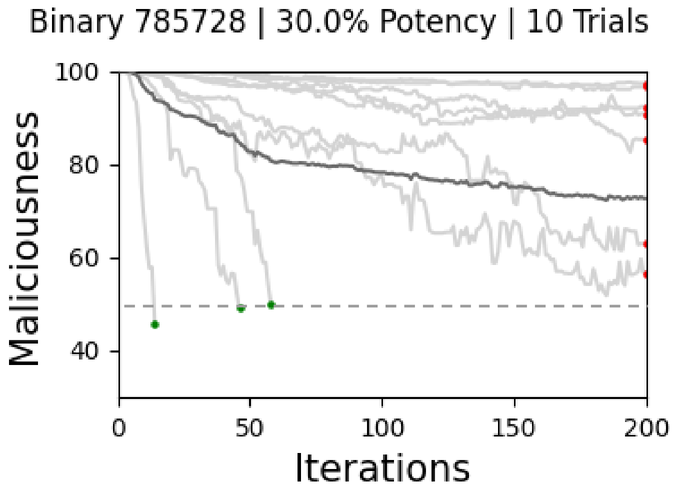
Attack behavior varies on a single binary

IPR attacks against Endgame

Results – Attack Behavior

Attack behavior varies on a single binary

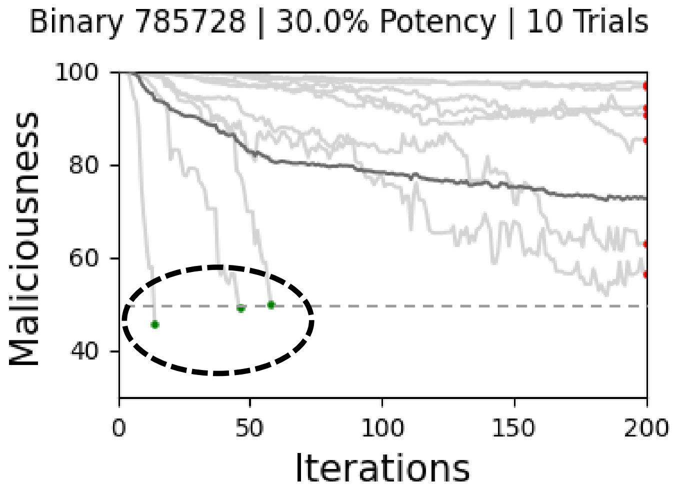
IPR attacks against Endgame



Results – Attack Behavior

Attack behavior varies on a single binary

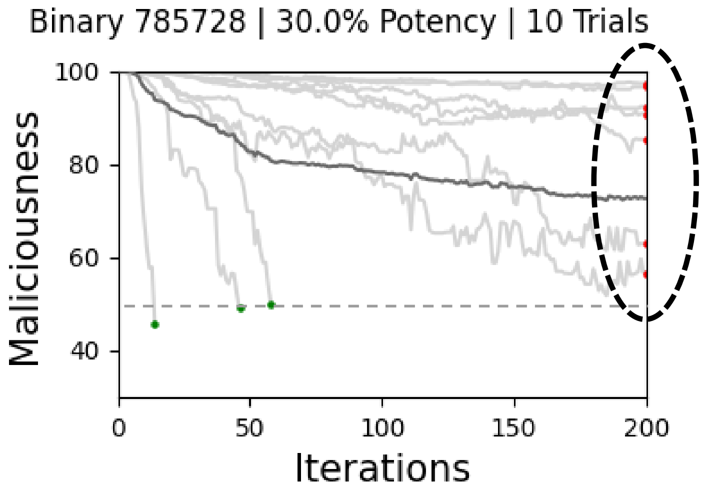
IPR attacks against Endgame



Results – Attack Behavior

Attack behavior varies on a single binary

IPR attacks against Endgame

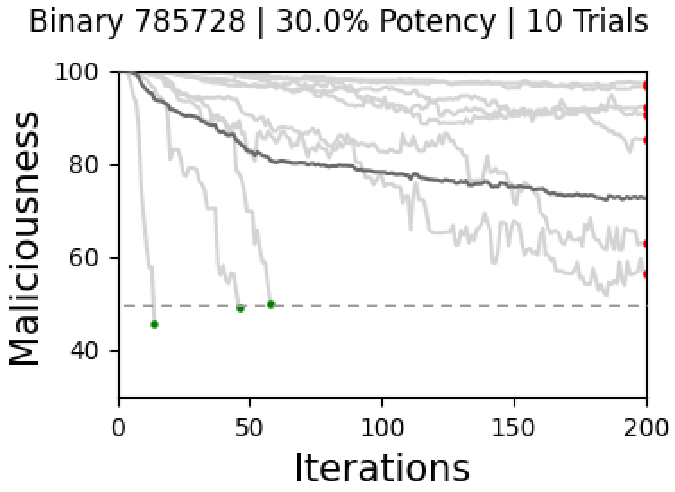


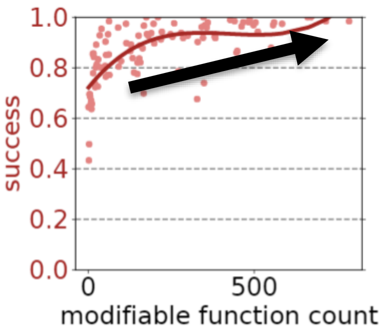
Results – Attack Behavior

Attack behavior varies on a single binary

Attack behavior varies between different binaries, depending on many variables

IPR attacks against Endgame



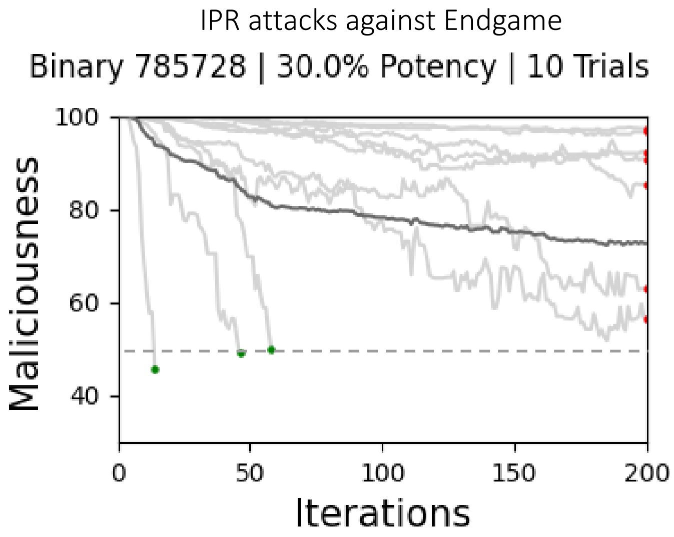


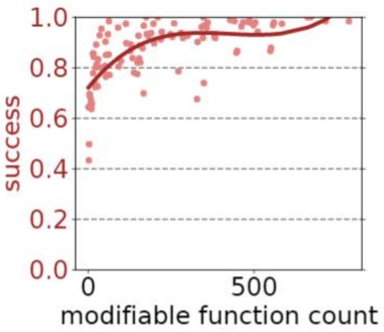
(a) Success (all attacks)

Results – Attack Behavior

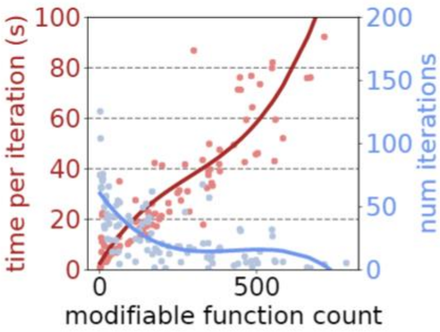
Attack behavior varies on a single binary

Attack behavior varies between different binaries, depending on many variables

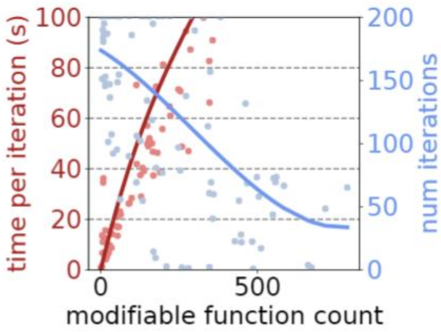




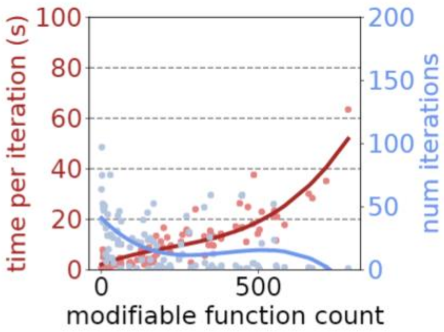
(a) **Success (all attacks)**



(b) **Time (all attacks)**



(c) **Time (*IPR*)**

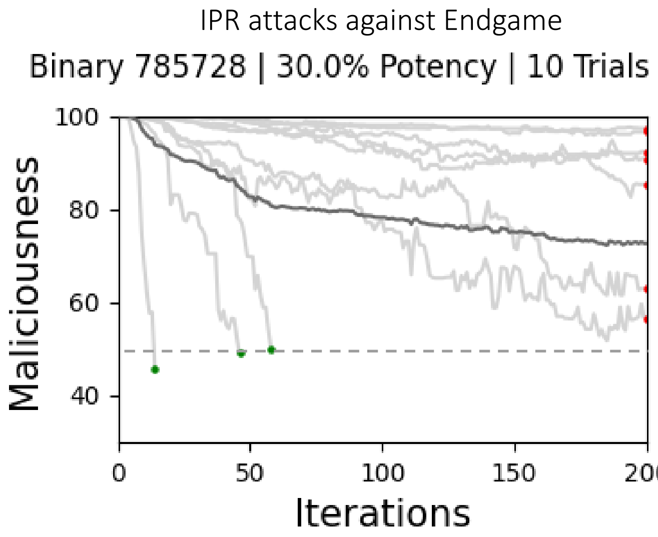


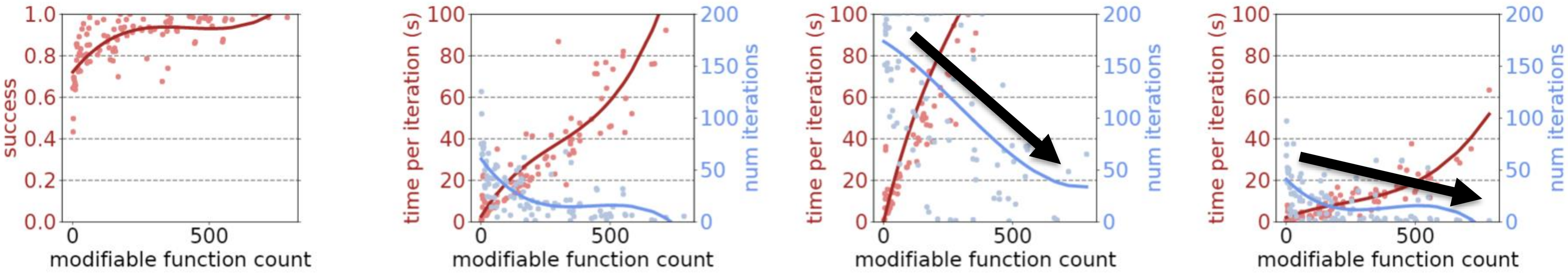
(d) **Time (*Disp*)**

Results – Attack Behavior

Attack behavior varies on a single binary

Attack behavior varies between different binaries, depending on many variables





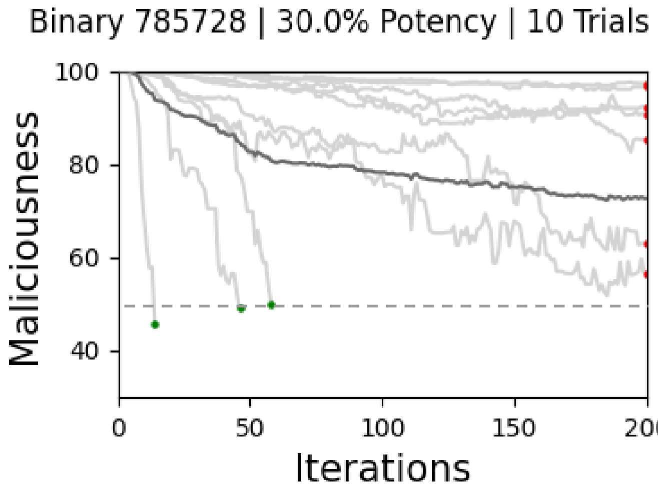
(a) Success (all attacks) (b) Time (all attacks) (c) Time (*IPR*) (d) Time (*Disp*)

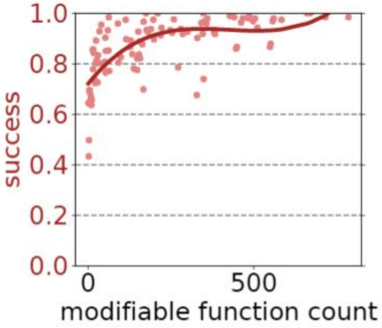
Results – Attack Behavior

Attack behavior varies on a single binary

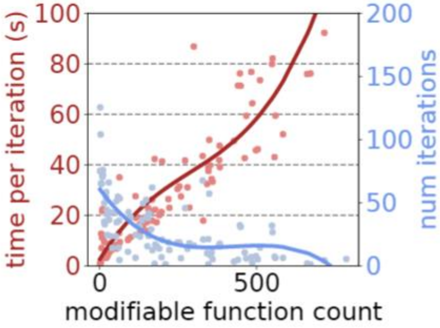
Attack behavior varies between different binaries, depending on many variables

IPR attacks against Endgame

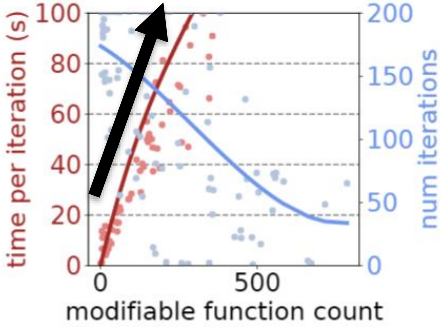




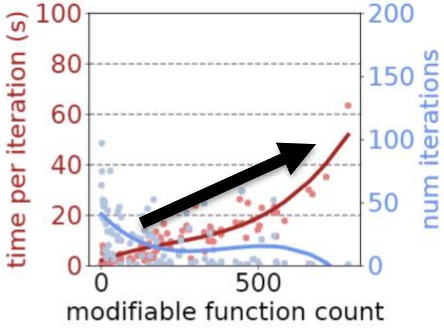
(a) Success (all attacks)



(b) Time (all attacks)

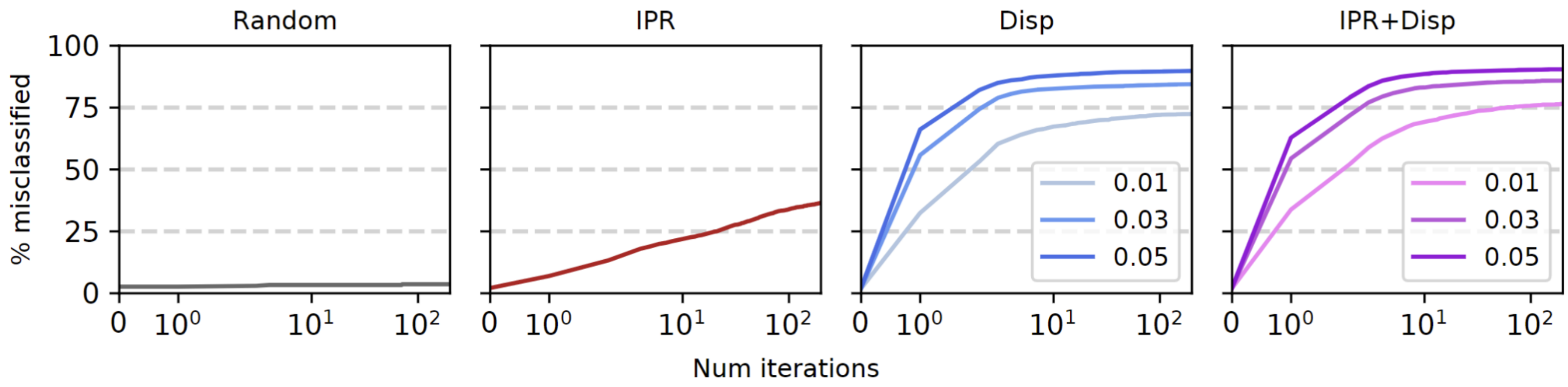


(c) Time (*IPR*)



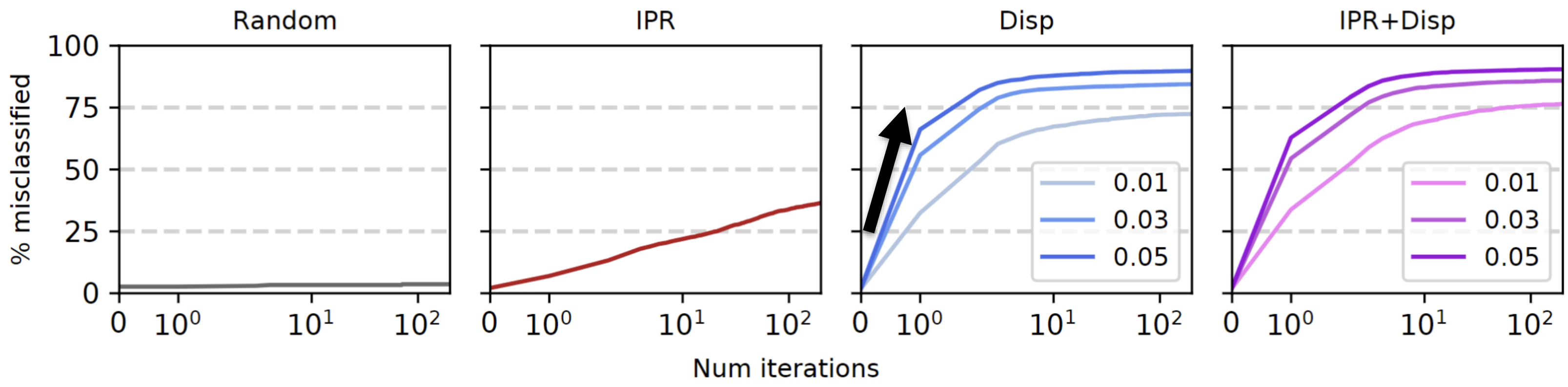
(d) Time (*Disp*)

Results – Contrasting Attack Types



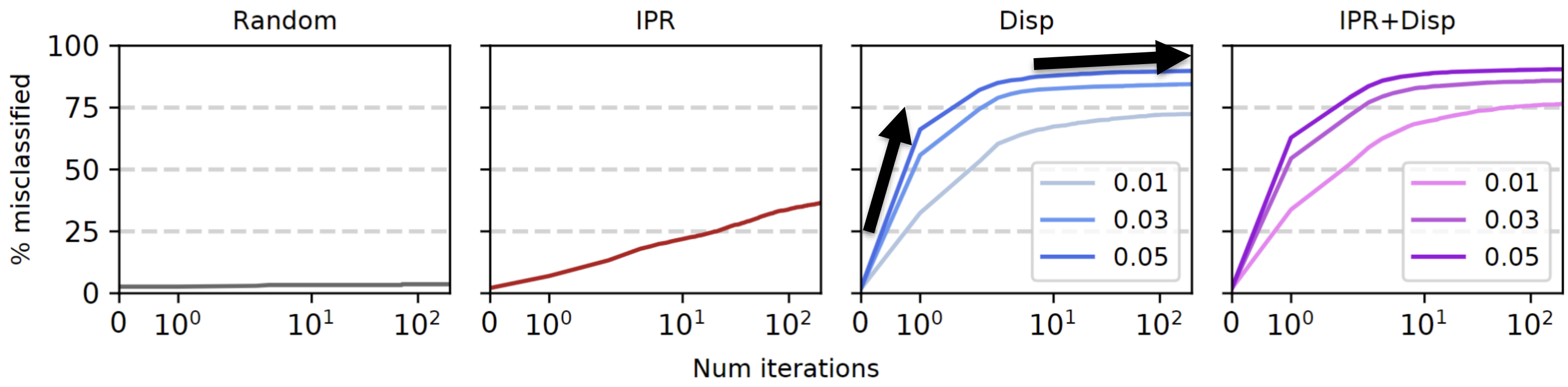
Attack success rates at each iteration in the white-box setting averaged over all target models and attacked binaries

Results – Contrasting Attack Types



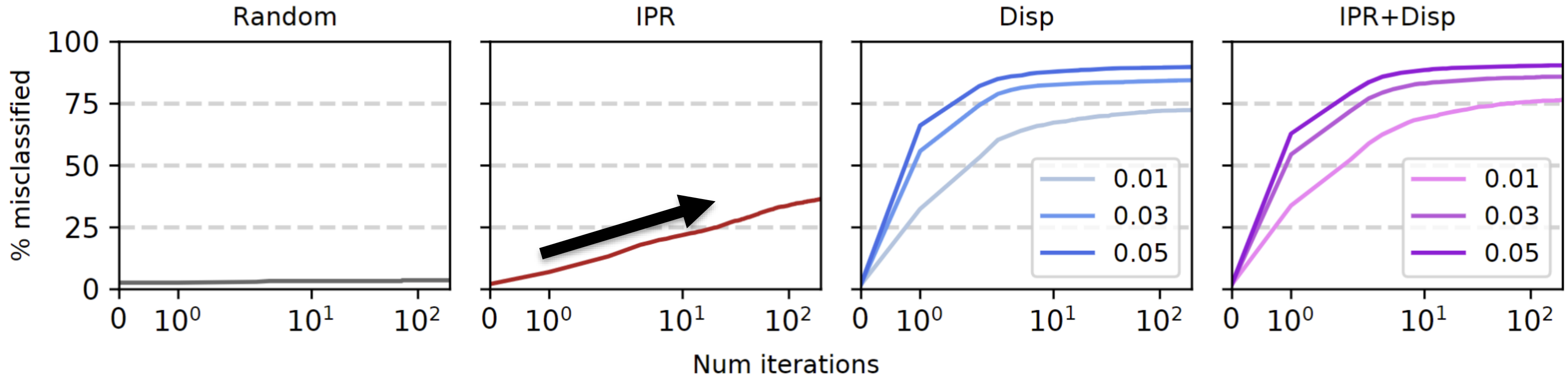
Attack success rates at each iteration in the white-box setting averaged over all target models and attacked binaries

Results – Contrasting Attack Types



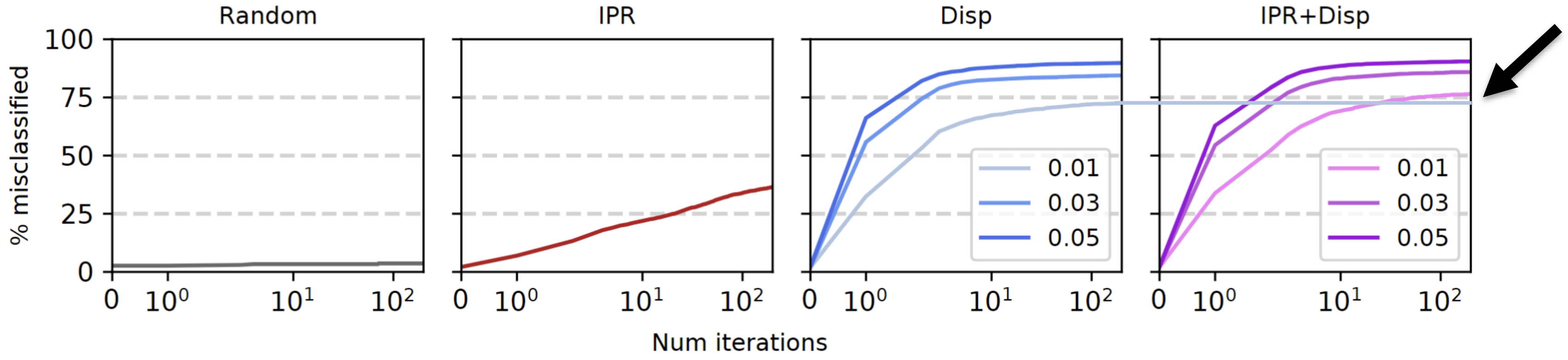
Attack success rates at each iteration in the white-box setting averaged over all target models and attacked binaries

Results – Contrasting Attack Types



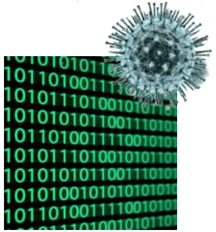
Attack success rates at each iteration in the white-box setting averaged over all target models and attacked binaries

Results – Contrasting Attack Types



Attack success rates at each iteration in the white-box setting averaged over all target models and attacked binaries

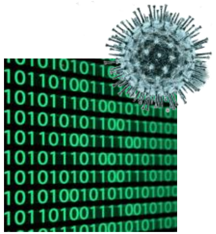
Results – Effects on Anti-Viruses



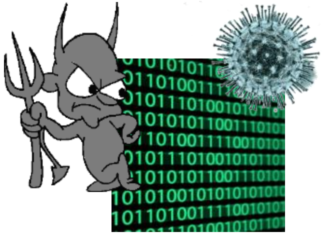
Unmodified malicious binaries were detected by a median of 55/68 AVs

VirusTotal. https://www.virustotal.com/. Online

Results – Effects on Anti-Viruses



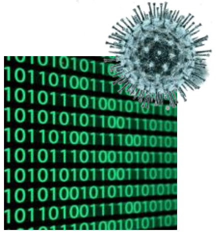
Unmodified malicious binaries were detected by a median of 55/68 AVs



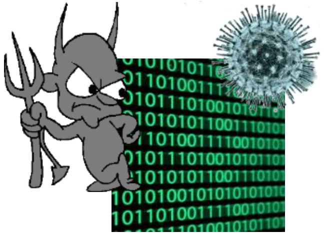
Randomly transformed malicious binaries were detected by a median of 42/68 AVs

VirusTotal. https://www.virustotal.com/. Online

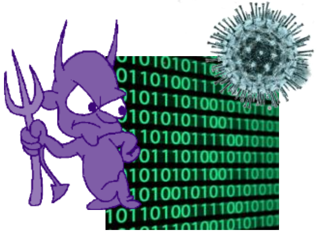
Results – Effects on Anti-Viruses



Unmodified malicious binaries were detected by a median of 55/68 AVs



Randomly transformed malicious binaries were detected by a median of 42/68 AVs



Adversarially transformed malicious binaries were detected by a median of 33-36/68 AVs

VirusTotal. https://www.virustotal.com/. Online

Potential Defenses

- Binary normalization – effective against IPR, ineffective against Displacement

Potential Defenses

- Binary normalization – effective against IPR, ineffective against Displacement
- Masking random instructions – effective when masking over 25% of instructions

Potential Defenses

- Binary normalization – effective against IPR, ineffective against Displacement
- Masking random instructions – effective when masking over 25% of instructions
- Adversarial training – currently not computationally feasible

Summary

- Described a process for modifying executable bytes of a binary to produce adversarial examples
  - Best attack succeeded in evading detection from all malware classification DNNs on nearly every binary
- Functionally preserving transformation code available on Github
  - Does not contain attack algorithm
  - https://github.com/pwwl/enhanced-binary-diversification
- Thank you for your time!

Malware Makeover: Breaking ML-based Static Analysis by Modifying Executable Bytes

Keane Lucas, Mahmood Sharif, Lujo Bauer, Michael K. Reiter, Saurabh Shintre