NEW STATIC ANALYSIS TECHNIQUES TO DETECT ENTROPY FAILURE VULNERABILITIES

Andrew Russell & Rushi Shah

December 13, 2018

The University of Texas at Austin

ENTROPY FAILURES IN THE WILD

- Debian OpenSSL (2008)
    - Removed key seed function call due to Valgrind error

ENTROPY FAILURES IN THE WILD

- Debian OpenSSL (2008)
    - Removed key seed function call due to Valgrind error
- FreeBSD (2016)
    - Kernel randomness never switches to “secure mode” after boot

TOWARD PROTECTING INDUSTRIAL SOFTWARE PROJECTS

- Modern software is large and iterative:
  - Code audits cost too much $$
- Idea: Use version control history to get better answers
- Audit code once, then use static analysis to prove you haven't introduced bugs with small changes to program
  - Key idea: Prove version 2 is secure relative to version 1.

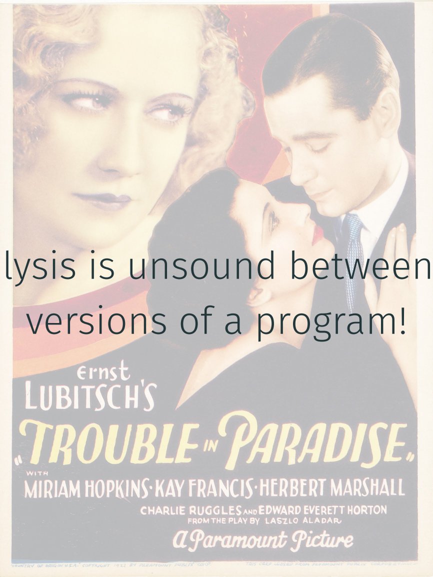
PROBLEM STATEMENT

Given two versions P_1, P_2 of a program, prove that if strong sources of entropy in P_1 flow into crypto algorithms, then the same is true of P_2.

If τ_1 is the taint set of variable x passed to sink in P_1 and τ_2 is the taint set of x in P_2, then we would like

$$\tau_1 == \tau_2$$

TROUBLE IN ANALYSIA



Taint analysis is unsound between multiple versions of a program!

OUR CONTRIBUTION

We propose a novel algorithm to detect relative entropy failures in a sound, tractable way.

SAFETY & STATIC ANALYSIS

- 2-safety property: making an assertion based on two runs of a program (ex: symmetry — $P(x, y) = P(y, x)$)
- Static analysis technique called "predicate abstraction" can prove 1-safety properties
- Existing techniques can reduce 2-safety properties into 1-safety properties via "product programs"

- Consider our symmetry example:

$$a := P(x, y); b := P(y, x); \text{assert}(a == b)$$

- However, it's hard to prove things about these sequentially composed programs

PRODUCT PROGRAMS

- Consider our symmetry example:

$$a := P(x, y); b := P(y, x); \mathrm{assert}(a == b)$$

- However, it's hard to prove things about these sequentially composed programs
- Instead, form product program $P_1 \times P_2 \equiv P_1 \; ; \; P_2$
    - "Synchronized" in a way that makes things easier for the verifier

PRODUCT PROGRAMS

P_1	P_2
if (p): x $\leftarrow$ 1; else: x $\leftarrow$ 2;	if (p): x $\leftarrow$ 2; else: x $\leftarrow$ 1;

PRODUCT PROGRAMS

$$P_1 \; ; \; P_2$$

---

if p **then**
    $x_1 \leftarrow 1$
else
    $x_1 \leftarrow 2$
end if
if p **then**
    $x_2 \leftarrow 2$
else
    $x_2 \leftarrow 1$
end if
assert($x_1 == x_2$)

$$P_1 \times P_2$$

```
if p then
    x₁ ← 1
    x₂ ← 2
else
    x₁ ← 2
    x₂ ← 1
end if
assert(x₁ == x₂)
```

APPROACH: HIGH-LEVEL OVERVIEW

1. Instrumentation: Convert taint analysis problem to safety problem
2. Product Program: Convert 2-safety property into 1-safety property
3. Use off-the-shelf static analysis tool to verify resulting program

$$
\begin{array}{llcl}
\textit{Statement} & S & := & \text{Atom} \\
 & & & \mid S_1 \,;\, S_2 \\
 & & & \mid \text{if } p \text{ then } S_1 \text{ else } S_2 \\
 & & & \mid \text{while } p\ S \\
\textit{Predicate} & p & := & \top \mid \bot \mid \text{Atom} \mid \neg p \mid p \odot p \\
\textit{Operator} & \odot & := & \wedge \mid \vee
\end{array}
$$

SYNCHRONIZED PRODUCT PROGRAM

- Basic inference rules:

$$\frac{}{A_1 \otimes A_2 \rightsquigarrow A_1 \ ; \ A_2}$$

SYNCHRONIZED PRODUCT PROGRAM

- Basic inference rules:

$$\frac{}{A_1 \otimes A_2 \rightsquigarrow A_1 \,;\, A_2}$$

$$\frac{S_2 \otimes S_1 \rightsquigarrow P}{S_1 \otimes S_2 \rightsquigarrow P}$$

SYNCHRONIZED PRODUCT PROGRAM

- Basic inference rules:

$$\frac{}{A_1 \otimes A_2 \rightsquigarrow A_1 \ ; \ A_2}$$

$$\frac{S_2 \otimes S_1 \rightsquigarrow P}{S_1 \otimes S_2 \rightsquigarrow P}$$

$$\frac{S_1 \otimes S \rightsquigarrow S_1' \quad S_2 \otimes S \rightsquigarrow S_2' \quad P = \mathit{if}(p) \ \mathit{then} \ S_1' \ \mathit{else} \ S_2'}{\mathit{if}(p) \ \mathit{then} \ S_1 \ \mathit{else} \ S_2 \otimes S \rightsquigarrow P}$$

SYNCHRONIZED PRODUCT PROGRAM

- Basic inference rules:

$$\frac{}{A_1 \otimes A_2 \rightsquigarrow A_1 \ ; \ A_2}$$

$$\frac{S_2 \otimes S_1 \rightsquigarrow P}{S_1 \otimes S_2 \rightsquigarrow P}$$

$$\frac{S_1 \otimes S \rightsquigarrow S_1' \quad S_2 \otimes S \rightsquigarrow S_2' \quad P = \mathit{if}(p) \ \mathit{then} \ S_1' \ \mathit{else} \ S_2'}{\mathit{if}(p) \ \mathit{then} \ S_1 \ \mathit{else} \ S_2 \otimes S \rightsquigarrow P}$$

$$\frac{P_0 = \mathit{while}(p_1 \wedge p_2) \ S_1 \otimes S_2 \quad P_1 = \mathit{while}(p_1) \ S_1 \quad P_2 = \mathit{while}(p_2) \ S_2}{\mathit{while}(p_1) \ S_1 \otimes \mathit{while}(p_2) \ S_2 \rightsquigarrow P_0 \ ; \ P_1 \ ; \ P_2}$$

SYNCHRONIZED PRODUCT PROGRAM

$$P_1 \times P_2$$

```
if p then
    x₁ ← 1
    if (p) x₂ ← 2 else x₂ ← 1 // P₂
else
    x₁ ← 2
    if (p) x₂ ← 2 else x₂ ← 1 // P₂
end if
assert(x₁ == x₂)
```

HYBRID PRODUCT PROGRAM

- Product program easiest to reason about for a verifier
- However: exponential blowup (in number of branches)

HYBRID PRODUCT PROGRAM

- Product program easiest to reason about for a verifier
- However: exponential blowup (in number of branches)
- Key insight: don't reason precisely about unrelated parts of the program
- "Unrelated" if not tainted. Use environment Γ (from instrumentation step) and add following inference rule:

$$\frac{\begin{array}{c}\Gamma \nvdash S_1 \\ \Gamma \nvdash S_2\end{array}}{\Gamma \vdash S_1 \otimes S_2 \rightsquigarrow S_1 \ ; \ S_2}$$

INSTRUMENTATION

- Replace sources with labelled constants
- For values that are tainted by more than one source (for example $S_1 + S_2$) replace with one of two uninterpreted functions over the sources:
    1. preserving$(s_1, s_2, \ldots, s_n)$. Ex: $+, \oplus$
    2. non-preserving$(s_1, s_2, \ldots, s_n)$. Ex: $<<, >>$
- Perform taint analysis on sources to generate environment Γ which marks statements involving tainted variables.

TAINT $\rightarrow$ SAFETY

For every variable x that is tainted in a statement s that is marked as a sink, insert:

$$\text{assert}(x_1 == x_2)$$

Recall we replaced sources with labelled constants and propagated them, so this will be asserting the taintsets of the two variables are equivalent.

TAINT $\rightarrow$ SAFETY

For every variable x that is tainted in a statement s that is marked as a sink, insert:

$$\text{assert}(x_1 == x_2)$$

Recall we replaced sources with labelled constants and propagated them, so this will be asserting the taintsets of the two variables are equivalent.

- If the assertion can be statically verified, P_2 is correct modulo P_1

CONCLUSION

- Progressive reduction from an unsound solution to a sound and tractable algorithm to solve entropy failure bugs in enterprise-scale software.
- Key insights: use version history, don't examine irrelevant parts of the program.

FIN

Thank you!

(especially Prof. Hovav Shacham and Prof. Işil Dillig)

Questions?