CS380S: Project Update

October 29, 2018

Rushi Shah Andrew Russell

One common misuse of cryptography is the misuse of entropy. Without proper random inputs, many cryptographic algorithms are vulnerable to basic forms of cryptanalysis. Some cryptographic schemes, such as DSA, can even disclose long-term secrets like the signing key when the random per-message input is low entropy, made public, or nonunique.

Given two versions P_1, P_2 of the same program P, we propose a relational verification for the entropy of the cryptographic values in the program. By introducing the concept of "differential taint analysis", we hope to produce a product program $P_1 \times P_2$ that is semantically equivalent to the sequential composition $P_1; P_2$, but such that we can prove useful safety properties of $P_1 \times P_2$ that would be difficult to prove with standard techniques on P_1, P_2, or $P_1; P_2$.

1 Motivating Example

Consider the following two programs P_1 and P_2 that are equivalent from a cryptographic point of view, but might have small feature changes.

function P_1
    $S \leftarrow$ entropy
    ⋮
    if f() **then**
        $c \leftarrow$ AES(s)
    end if
    ⋮
end function

function P_2
    $S \leftarrow$ entropy
    ⋮
    if f() **then**
        $c \leftarrow$ AES(s)
    end if
    ⋮
end function

where f is some complex function that is difficult to reason about, but does not change between P_1 and P_2. Then, safety properties about the entropy of the cryptographic values in c would hold in both programs or in neither program.

2 Existing Techniques

Static code analysis has a history of identifying security vulnerabilities at a source code level. Some examples include SQL injection, cross-site scripting exploits, and buffer overflow attacks. However, there has not been any attempt in the literature to statically analyze source code for cryptographic vulnerabilities stemming from entropy misuse, which we seek to do. Standard static code analysis techniques developed thus far are also insufficient for the analysis we would like to perform.

Under-approx tools don't exist (why do we need under approx tools?), and also difficult to reason about arbitrary functions (but they are usually irrelevant to the analysis you're performing), also can't just run taint analysis on one program because don't know what the taint results should look like (the old program gives us a correctness criteria/oracle to infer what the correct taint results would be).

3 Our Approach

Introduce concept of differential taint analysis here. Talk about constructing a product program.

function $P_1 \times P_2$
    $S_1 \leftarrow$ entropy
    $S_2 \leftarrow$ entropy
    ⋮
    if f() **then**

$c_1 \leftarrow \text{AES}(s_1)$
$c_2 \leftarrow \text{AES}(s_2)$
end if
⋮
end function

Note that, during the construction of the product program, we need to be able to merge the if statements in the two programs. Also note that we need to demonstrate set equality between the taint for s_1, s_2 when they are used in the if statement.

4 What We Hope To Show

Talk about how we assign the output of a sufficiently entropic value to a variable in P_1 iff we also assign it in P_2. Then show that there is an equivalence in the taint propagation (prove set equality of the places where cryptographic values are used).

5 Real World Applications

How our analysis framework would be able to identify real world bugs. What are the benchmarks we compare against (OpenSSL, FreeBSD, etc.).

6 Research Hypotheses

These are the principal hypotheses we would like to test:

1. An automated tool can detect entropy bugs in real-world programs.
2. Entropy is insufficiently propagated in programs that rely on cryptography.
3. Multiple versions of the same program can make static analysis for this domain more effective

7 Links

1. Debian/OpenSSL Bug
   (a) https://www.schneier.com/blog/archives/2008/05/random_number_b.html
   (b) https://research.swtch.com/openssl
   (c) https://freedom-to-tinker.com/2013/09/20/software-transparency-debian-openssl-bug/
   (d) https://www.cs.umd.edu/class/fall2017/cmsc818O/papers/private-keys-public.pdf
2. Data flow
   (a) https://en.wikipedia.org/wiki/Data-flow_analysis
   (b) https://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec02-Dataflow.pdf
3. Static Program Analysis
   (a) https://cs.au.dk/~amoeller/spa/spa.pdf
   (b) https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6859783
4. Relational Verification:
   (a) https://dl.acm.org/citation.cfm?id=2021319
   (b) https://ac.els-cdn.com/S235222081630044X/1-s2.0-S235222081630044X-main.pdf?_tid=076a0492-9cee-4995-9710-bcb3c64b98e0&acdnat=1539815890_178849b4f14af3751e9acb03b238db4d
   (c) https://www.microsoft.com/en-us/research/publication/differential-assertion-checking/
   (d) https://www.microsoft.com/en-us/research/wp-content/uploads/2014/06/paper-1.pdf
   (e) https://www.cs.utexas.edu/~isil/pldi16-chl.pdf
5. Projects to analyze
   (a) OpenPGP
   (b) BouncyCastle
   (c) OpenSSL
   (d) GnuPGP
   (e) F# SSL project with proof of correctness
   (f) NQSBTLS
   (g) Amazon's s2n (signal to noise)