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 $P_1; P_2$. The safety property we are the most interested in proving is that cryptographic values rely on sufficiently many bits of entropy when they are used.

1 Example Formalization

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, clearly, our safety properties about the entropy of the cryptographic values in c would hold in both programs or in neither program.

2 Existing Techniques, DTA Motivation

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.

Standard taint analysis will under/overapproximate the taint, so we may not be able to use it to directly prove our safety property on the target program. Thus, we can leverage the approach of relational verification.

Relational verification aims to solve this problem by proving relational properties between two programs. However, the naive relational verification approach of verifying each program individually is infeasible given arbitrarily difficult-to-reason-about functions even if those functions do not change between versions of the program. One approach is to underapproximate the taint on one program and overapproximate the taint on another program to verify the set equality of taint sources, but this still falls prey to the approximation of taint analysis given arbitrarily complex functions. Also, current tooling does not widely support underapproximating taint analysis.

Finally, even with a sufficiently precise taint analysis,

it is unclear how to determine the "correct" set of sources a cryptographic value should rely on at any point in the program. For example, some programs will use weak entropy on system startup, and increase the entropy in those values, later on. Thus, by running our analysis on two versions of the program, we can use one version as an oracle for the correctness criteria of the other program. In other words, we can use it to infer what the set of taint results should be for any variable at any point in the program.

3 Our Approach

Given the two programs in the example, we propose generating a product program similar to the following:

function $P_1 \times P_2$
  $S_1 \leftarrow$ entropy
  $S_2 \leftarrow$ entropy
  $\vdots$
  if f() **then**
    $c_1 \leftarrow \text{AES}(s_1)$
    $c_2 \leftarrow \text{AES}(s_2)$
  end if
  $\vdots$
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.

We would like to demonstrate set equality between the taint for s_1, s_2 when they are used in the if statement, which amounts to demonstrating the equivalence of the taint propagation through the program. We would also like to demonstrate that we assign the output of a sufficiently entropic value to a variable in P_1 if and only if we also assign it that sufficiently entropic value in P_2.

Proving these two properties (set-equality and equivalence of assignments) allows us to claim that P_1 has the safety property if and only if P_2 has the safety property.

4 Real World Applications

A verification tool like this could be especially useful for a maintainer of a project that relies on cryptographic values. For example, if she audits her code once to confirm that it uses entropy properly, she can use differential taint analysis on future commits to confirm that the new code does not introduce this class of bugs.

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

5 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

6 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)