New Static Analysis Techniques to Detect Entropy Failure Vulnerabilities in Modern Software Projects

December 13, 2018

Rushi Shah Andrew Russell

Abstract

Test abstract

1 Introduction

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 such as the signing key when the random per-message input is low entropy, made public, or nonunique.

We plan to use data dependency tools to determine how entropic inputs in a given program are used by various cryptographic algorithms. That will allow us to identify if and when entropy is too low or is misused. We plan to produce this as a code integration tool for developers to use as part of a compiler toolchain. Our tool will seek to generate an error report from source code using notions of taint analysis.

For this project we consider codebases that maintain version history, in the hopes that we can employ static analysis techniques across program versions to infer more information about how entropy is being used; our primary goal is to identify bugs that are introduced to existing codebases. This may prove to be especially helpful since naive taint analysis can provide mere over-approximations of data dependency for a given program and may generate many false alarms for programmers in a professional development setting.

1.1 Related Work

Differential assertion checking, verification modulo versions, relational verification using product programs, secure information flow as a safety problem

Figure 1:

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

2 Preliminaries

2.1 Demonstrative Language

We will build our tool for a language consisting of a small set of semantic rules (Figure 1).

Our results can be extended in a straightforward manner to an industrial language such as C, and we do so for our evaluation (Section 4).

2.2 Taint Analysis

Taint analysis is a standard program analysis technique, typically used to detect security vulnerabilities in either a static or dynamic manner. We restrict our focus to the static variant. We define a taint analysis algorithm as follows:

- $\Gamma \leftarrow \mathsf{Taint}_{S,L}(\phi, T, \Sigma, Z, P)$: Takes as input an initial assignment ϕ of some taint sets $T = \{\tau_i\}$ to a set of sources $\Sigma = \{\sigma_i\}$, and an input program P. Outputs an assignment Γ of statements s_i in the program P to taint sets $\tau_i \in T$ according to the taint propagation semantics S for programming language L.

An example of static taint analysis is to determine if unsanitized user inputs are ever provided to vulnerable functions, such as SQL commands or a webpage templating function. In our case we wish to ensure that entropy-sensitive inputs to cryptographic functions (sinks) can be traced back to a high-entropy source such as *nix's `/dev/urandom`.

2.3 k-safety properties

As opposed to a safety property of a program (for example, "variable x is always positive" or "pointer p is never null"), which requires the absence of errors in a single program trace [1], a k-safety property of a program requires the absence of erroneous interactions between k traces of the same program [5]. A program property such as symmetry is a 2-safety property.

2.4 Predicate Abstraction

Predicate abstraction is a specialized form of abstract interpretation that can be used for checking 1-safety properties of programs [4]. Abstract interpretation is a static analysis technique for programs that allows a verifier to, in some cases, automatically infer properties such as loop invariants. Formally we have:

- $b \leftarrow \mathsf{Verify}_L(\pi, P)$: Takes as input a 1-safety property π and a program P and outputs an accepting bit $\top$ if P satisfies the safety property, and otherwise outputs a rejecting bit $\perp$ or fails to terminate otherwise.

CPAchecker [3] is an example of a state-of-the-art predicate abstraction verifier for the C programming language. In this work we are interested in the safety property of equality. We note that due to inherent hardness results such as NP-hardness and undecidability, predicate abstraction verifiers are based on heuristic methods and are not guaranteed to run in polynomial time, or even terminate.

2.5 Product Programs

The product $P_1 \times P_2$ of two programs P_1 and P_2 is not an entirely well-defined concept. In the most general sense, the product $P_1 \times P_2$ of two programs P_1 and P_2 is used to verify relations between the programs, such as equivalence [2]. Product programs have also been used to analyze different runs of the same program. The product program $P_1 \times P_2$ 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 $P_1;P_2$.

For our purposes we simply require that $P_1 \times P_2$ is semantically equivalent to the sequential composition $P_1 \; ; \; P_2$ of the two programs. Our goal in using the concept of product programs is to aid the predicate abstraction verifier.

Figure 2: Taint analysis overapproximation

if $f()$ **then**
  $k \leftarrow \mathsf{read}(128,\ \text{"/dev/urandom"})$
else
  $k \leftarrow \mathsf{getpid}()$
end if
$\mathsf{ctxt} \leftarrow \mathsf{AES}(k, \mathsf{ptxt})$

3 Our Approach

Our main approach to this problem is to take two versions P_1 and P_2 of a program and prove P_2 *relative* to P_1. We have several motivations for this approach:

- Taint analysis is an (over)approximate solution.
- Modern software projects often use version control history, and we can leverage that to give a static verifier more power (heuristically speaking).

Let us address the first point. An initial approach to this problem might consist of using off-the-shelf taint analysis (we would taint high-entropy sources and confirm that cryptographic functions receive high-entropy tainted inputs). However, taint analysis merely provides an overapproximation to the actual entropy that is input to a cryptographic algorithm. Consider the program in Figure 2. The variable k is tainted with high-entropy, but will not contain high-entropy if $f()$ ever evaluates to false.

Now we consider the second point. Since software developers often have access to multiple revisions of a single program, we wish to use this additional information to do better than simple taint analysis on a single version. This allows for stronger security guarantees at lower cost: a software project can invest in manually auditing an initial version of its security-critical software, and then use our methods to prove that subsequent versions do not contain entropy failures provided that this initial version does not (i.e., a *relative* or *differential* notion of security). Additionally, since we expect program revisions (particularly those touching security-critical components) to be infrequent and small, we can reduce the scope of the verifier's work.

Figure 3: Taint analysis unsoundness

if $f()$ **then**
  $k \leftarrow \mathsf{getpid}()$
else
  $k \leftarrow \mathsf{read}(128,\ \text{"/dev/urandom"})$
end if
$\mathsf{ctxt} \leftarrow \mathsf{AES}(k, \mathsf{ptxt})$

3.1 A First Attempt

Given that we wish to attain this differential notion of security, we first address a naive attempt at solving this problem using taint analysis. Consider the program in Figure 3 compared to the program in Figure 2. The taint sets of k in both versions of the program are equal, however, simply comparing for equality here is *unsound* with respect to our differential notion of security, since for all program traces in which the first program is secure, all similar program traces in the second program are insecure.

3.2 Differential Taint Analysis

Description of differential taint analysis.

3.3 Overview

Our solution proceeds in three primary stages:

- **Instrumentation:** In this step we reduce our taint analysis problem for each program version P_1 and P_2 to one of 2-safety. Additionally, we run taint analysis on each program to get outputs Γ_1 and Γ_2, which we use in the next step.
- **Program syncronization:** Using our environment $\Gamma = \Gamma_1 \cup \Gamma_2$, our instrumented programs P_1' and P_2' from the previous step, we construct the product program $P_1' \times P_2'$. This converts the 2-safety problem to one of 1-safety.
- **Verification:** Finally, we pass $P_1' \times P_2'$ to an off-the-shelf verifier to check the 1-safety property to prove security.

We describe these in more formal detail in the coming sections.

3.4 Instrumenting Programs for Predicate Abstraction

Instrumentation: Given two program versions P_1 and P_2, our sources Σ, and a taint analysis algorithm $\mathsf{Taint}_{S,L}$

Figure 4: Basic Inference Rules

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

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

Figure 5: Conditional and Loop Inference Rules

$$\frac{\begin{array}{l} S_1 \otimes S \rightsquigarrow S_1' \\ S_2 \otimes S \rightsquigarrow S_2' \\ P = if(p)\ then\ S_1'\ else\ S_2' \end{array}}{if(p)\ then\ S_1\ else\ S_2 \otimes S \rightsquigarrow P}$$

$$\frac{\begin{array}{l} P_0 = while(p_1 \wedge p_2)\ S_1 \ ; \ S_2 \\ P_1 = while(p_1)\ S_1 \\ P_2 = while(p_2)\ S_2 \end{array}}{while(p_1)\ S_1 \otimes while(p_2)\ S_2 \rightsquigarrow P_0 \ ; \ P_1 \ ; \ P_2}$$

we reduce the taint analysis problem to one of 1-safety. We do this for each P_k by assigning constants c to all sources $\sigma \in \Sigma$, and then where we would propagate taint using the rules S for language L in statements s_i which is a function of program variables v_i and only of "entropy preserving" operations (described below) we replace with an uninterpreted function $f(v_1, \ldots, v_j)$. For statements s_i' which are a function of any tainted program variables w_j and contain any "non-entropy preserving operations" (any operations that are not entropy preserving), we replace with an uninterpreted function $g(w_1, \ldots, w_j)$. We output these programs as P_1' and P_2'.

Description of instrumentation algorithm.

3.5 Naive Product Program Inference Rules

Description of product program inference rules approach.

3.6 Taint-augmented Inference Rules

Why the naive approach doesn't work (intractability). Introduce heuristic-based approach.

3.7 Putting it all together

Description of algorithm using the two subroutines above.

Figure 6: Augmented Inference Rules

$$\frac{\Gamma \nvdash S_1 \quad \Gamma \nvdash S_2}{\Gamma \vdash S_1 \otimes S_2 \rightsquigarrow S_1 \ ; \ S_2}$$

Figure 7: Augmented Inference Rules

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

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

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

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

4 Evaluation

5 Conclusion

In this work we present a new static analysis technique to aid in the prevention of entropy-misuse security vulnerabilities, such as those found in the Debian OpenSSL and FreeBSD projects.

6 Future Work

Verify the following conjectures. Prove that a heuristic-based approach is no less powerful than a full product program approach. Run this tool on projects in the wild to determine how widespread entropy-misuse bugs are among software projects "in the wild."

7 Considered Approaches

7.1 Taint Analysis

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. We claim that standard static code analysis techniques developed thus far are insufficient for the analysis we would like to perform. The sources of entropy can be tainted standard taint analysis techniques can be used to approximate how the taint propagates. But this approximation may be too coarse to provide useful information to the user. 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.

7.2 Differential Taint Analysis

Since we want a stronger notion of taint analysis than a naive approach, we intend to introduce and formalize the concept of "differential taint analysis." Differential taint analysis, in the ideal, will give us more information than simple taint analysis by leveraging the original version of the program as a source of ground truth. This additional information could include reducing programmer annotation work or reduce false positives between versions (since taint analysis overapproximates data flow). By viewing the original version of the program as bug-free, we reduce the number of code audits required: only the original version of the code would need to be audited, and future versions could rely on our tool to automatically detect bugs that were introduced by refactors.

7.3 Product Programs

One technique that we have considered toward the construction of differential taint analysis is the notion of a "product program." In the most general sense, the product $P_1 \times P_2$ of two programs P_1 and P_2 is used to verify relations between the programs, such as equivalence. Product programs have also been used to analyze different runs of the same program. The product program $P_1 \times P_2$ 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 $P_1; P_2$.

In our setting, we are not so much interested in proving safety properties of a program P but instead wish to prove that cryptographic values rely on sufficiently many bits of entropy when they are used via taint analysis, which is why we cannot simply leverage existing off-the-shelf product program analysis techniques.

7.3.1 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 p **then**
        $c \leftarrow$ AES(s)
    end if
    ⋮
end function

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

where p is some complex predicate that is difficult to reason about, but does not change between P_1 and P_2 (and no values that p depends on change, either). Then, a safety property about the entropy of the cryptographic values in c would hold in both programs or in neither program.

7.4 Verification Modulo Versions

Verification modulo versions is an idea quite similar to product programs. VMV attempts to identify abstract regressions and relative correctness between two versions of a program. In VMV, the static analyzer is treated as opaque in an effort to cut down the number of alarms by inferring assumptions made by the original program (assumed to be correct). This is especially appealing in the security software development cycle because there are documented cases of entropy bugs being introduced in new versions of programs (for example, OpenSSL and FreeBSD). This would involve under-approximating the taint of one version of the program and over-approximating the taint of another version of the program. Current tooling does not widely support under-approximating taint analysis.

7.5 Relational Verification

Another existing technique is relational verification, in which relational properties between two programs are used to prove k-safety properties. A k-safety property is a safety property of a program that requires reasoning about the relationships between k different runs of a program (or k programs). For example, transitivity is a k-safety property. In our case, we would want to show the equivalence of entropy assignment and the equivalence of the entropy propagation. Relational verification utilitzes product programs; RV constructs a product program between versions of the program that captures the semantics of executing them sequentially, but that lends itself to standard verification techniques.

7.6 Differential Assertion Checking

The last approach that we have considered is differential assertion checking. Differential assertion checking seeks to (statically) prove the relative correctness between two similar programs, with a significantly lower cost than ensuring absolute correctness. That is, DAC allows is to answer if there are conditions under which a program P_1 passes an assertion check but a program P_2 fails. The techniques utilized current DAC work may be of interest, but it seems they rely on an a priori mapping between the two programs, which is something that relational verification work seeks to answer. Thus DAC will be considered further after the idea of product programs is exhausted.

8 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)

References

[1] ALPERN, B., AND SCHNEIDER, F. B. Recognizing safety and liveness. *Distributed computing 2*, 3 (1987), 117–126.

[2] BARTHE, G., CRESPO, J. M., AND KUNZ, C. Relational verification using product programs. In *International Symposium on Formal Methods* (2011), Springer, pp. 200–214.

[3] BEYER, D., AND KEREMOGLU, M. E. Cpachecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification* (2011), Springer, pp. 184–190.

[4] FLANAGAN, C., AND QADEER, S. Predicate abstraction for software verification. In *ACM SIGPLAN Notices* (2002), vol. 37, ACM, pp. 191–202.

[5] SOUSA, M., AND DILLIG, I. Cartesian hoare logic for verifying k-safety properties. In *ACM SIGPLAN Notices* (2016), vol. 51, ACM, pp. 57–69.