New Static Analysis Techniques to Detect Entropy Failure Vulnerabilities

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

2 Preliminaries

2.1 Toy Language

We will build our tool for a language consisting of a small set of semantic rules (Figure 2.1).

Figure 1:

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

Our results can be extended in a straightforward manner to an industrial standard language such as C.

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, a k-safety property of a program requires the absence of erroneous interactions between k traces of the same program [**?**]. 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 [**?**].

2.5 Product Programs

3 Our Approach

Want to use differential view since we can get more out of this. Unsound taint analysis example.

3.1 Differential Taint Analysis

Description of differential taint analysis.

3.2 Instrumenting Programs for Predicate Abstraction

In the instrumentation phase, we take an entire software project, and augment it with the other inputs (like annotations) to prepare it for the static analysis we will perform. Ultimately, our goal is to make assertions about the taintsets of program variables at the sinks. But first, we will individually instrument each version of the software project.

First we transform the code from its existing state into the language whose semantics we described earlier. We also rename the variables in the second version of the program so their names are disjoint from the variable names in program one. Then, in both programs, we replace the legitimate sources of entropy with labelled constants. We propogate these labelled constants through the program to propagate the taint of the entropy. In the process of doing so, it is possible that a value is tainted by more than one source (which is why we are tracking the taintset). We replace statements that taint a value with two or more sources $(s_1, s_2, \ldots, s_n)$ with one of two uninterpreted functions:

1. $\mathsf{preserving}(s_1, s_2, \ldots, s_n)$ if the operation preserves entropy (+, XOR, etc.)

Figure 2: Basic Inference Rules

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

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

2. $\mathsf{non\text{-}preserving}(s_1, s_2, \ldots, s_n)$ if the operation does not preserve entropy («, », etc.)

These uninterpreted functions are inferred to be pure functions over their variables, so they capture the notion of the taintset at that point. This allows us to assert their equality in the program we pass to CPAChecker.

Next, we will collect information from the two versions of the program as we transform them that will guide our future inference rules. Namely, we will perform taint analysis on sources to populate the environment Γ which marks statements involving tainted variables. $\Gamma \vdash S_1$ if S_1 references a variable that is tainted by a source with sufficiently high entropy.

3.3 Naive Product Program Inference Rules

With the instrumented programs, we can construct a product program that is semantically equivalent to the sequential composition of the two programs. However, we would like this product program to be more amenable to reason about for the static analysis tool. To this end, we would like to synchronize the control flow of the two programs as much as possible.

We present inference rules for composing statements in the two programs $S_1 \otimes S_1$ into one program that is semantically equivalent to $S_1 \; ; \; S_2$. The most basic inference rule simply involves sequentially composing two atomic statements, which do not involve any control flow to be syncrhonized. Also, because the variables in the two versions of the program have been renamed to be disjoint, we can introduce an inference rule for the commutativity of two statements.

Next, we have the inference rule for synchronizing an if-statement with another statement. If S_1 is the taken branch, S_2 is the not-taken branch, and S is the statement after the entire if-else-statement, then we would like to inject S separately into each branch. This simple transformation is clearly equivalent to the sequential composition of the if-statement and S. The benefit of this transformation, however, is that the predicate abstraction tool has more information about the trace taken with which it can reason about product program.

Figure 3: Conditional and Loop Inference Rules

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

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

Finally, we have the inference rule for synchronizing two while loops. The intuition here is that we would like the two loops to operate in lock-step as long as possible. To maintain the equivalence with the sequential composition, we also run the remaining iterations in the leftover loop after the other loop's condition no longer holds.

Although these inference rules are semantically equivalent to the sequential composition, and clearly provide benefit to CPAChecker, this proving strength comes at a cost. Namely, the if statements and the while loops lead to an exponential blowup in the size of the product program being reasoned about as a result of the copied statements.

3.4 Taint-augmented Inference Rules

We conjecture that the exponential blowup introduced by the naive inference rules is avoidable in some cases. In particular, our key insight is that we don't need the extra proving power provided by the synchronization in parts of the program that are unrelated to entropy. For these program statements, we can resort to the sequential composition. For example, consider a large software program: only a small portion of the code base will be related to the entropy. Therefore, we should concentrate the bulk of our analysis on this small portion of the code.

We define these "unrelated" statements to be the ones that are not tainted by any legitimate entropy source. This is reasonable because our ultimate assertions we would like to prove will only be over the tainted variables. Thus, we can use the Γ environment we constructed in the instrumentation step to augment our inference rules. We say that $\Gamma \vdash S$ if some variable referenced in S is tainted by at least one legitimate source of entropy. Similarly, $\Gamma \nvdash S$ if no variable in the statement is tainted by any legitimate source of entropy. We introduce an optimization inference rule that does not lead to any exponential blowup for statements that are not entailed by Γ.

Figure 4: Augmented Inference Rules

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

Figure 5: Augmented Inference Rules

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

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

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

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

$$\frac{\begin{array}{c} P_0 = while(p_1 \wedge p_2)\ S_1\ ;\ S_2 \\ P_1 = while(p_1)\ S_1 \\ P_2 = while(p_2)\ S_2 \\ P = if(p)\ then\ S_1'\ else\ S_2' \end{array}}{\Gamma \vdash while(p_1)\ S_1 \otimes while(p_2)\ S_2 \rightsquigarrow P_0\ ;\ P_1\ ;\ P_2}$$

Then, we can make the corresponsding changes to our naive inference rules to recognize the Γ environment we've introduced. This gives us the following set of augmented inference rules:

3.5 Putting it all together

Description of algorithm using the two subroutines above.

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.