University of California
Santa Barbara

Identifying Program Entropy Characteristics with Symbolic Execution

A Thesis submitted in partial satisfaction
of the requirements for the degree

Master of Science
in
Computer Science

by

Audrey Annika Dutcher

Committee in charge:

Professor Giovanni Vigna, Chair
Professor Christopher Kruegel
Professor Yan Shoshitaishvili

March 2019

The Thesis of Audrey Annika Dutcher is approved.

Professor Christopher Kruegel

Professor Yan Shoshitaishvili

Professor Giovanni Vigna, Committee Chair

March 2019

Identifying Program Entropy Characteristics with Symbolic Execution

Audrey Annika Dutcher

Acknowledgements

Many thanks to Chani Jindal, who worked with me on the class project out of which this thesis grew.

Many more thanks to Fish Wang, for being extremely supportive, sometimes in excess but always with love.

Abstract

Identifying Program Entropy Characteristics with Symbolic Execution

by

Audrey Annika Dutcher

The security infrastructure underpinning our society relies on encryption, which relies on the correct generation and use of pseudorandom data. Unfortunately, random data is deceptively hard to generate. Implementation problems in PRNGs and the incorrect usage of generated random data in cryptographic algorithms have led to many issues, including the infamous Debian OpenSSL bug, which exposed millions of systems on the internet to potential compromise due to a mistake that limited the source of randomness during key generation to have 2^{15} different seeds (i.e. 15 bits of entropy).

It is important to *automatically* identify if a given program applies a certain cryptographic algorithm or uses its random data correctly. This paper tackles the very first step of this problem by extracting an understanding of how a binary program generates or uses randomness. Specifically, we set the following problem: given a program (or a specific function), can we estimate bounds on the amount of randomness present in the program or function's output by determining bounds on the *entropy* of this output data? Our technique estimates upper bounds on the entropy of program output through a process of expression reinterpretation and stochastic probability estimation, related to abstract interpretation and model counting.

Contents

Chapter 1

Introduction

The security infrastructure underpinning our society relies on encryption. Encryption safeguards our personal data, protects our bank accounts, and secures our communications. The encryption schemes we use today rely on encryption keys that are used to encrypt and sign data. In turn, these encryption keys are created through the generation of random data.

Unfortunately, random data is deceptively hard to generate. Truly random data can be sampled from the environment, but this has low throughput using normal techniques (i.e., measuring radio interferences) or requires specialized equipment. As an alternative, *pseudorandom number generators* (PRNG) have been developed to provide high-throughput, pseudorandom data generation.

A high-quality true-random number generator (RNG) is almost always required for security, and lack of quality generally provides attack vulnerabilities and so leads to lack of security, even to complete compromise, in cryptographic systems. Implementation problems in PRNGs, and the incorrect usage of generated random data in cryptographic algorithms, have led to many such issues in the past. For example, the infamous Debian OpenSSL bug [1], which exposed millions of systems on the internet to potential compromise, was caused by a mistake in the binary distribution of Debian's openssl package that limited the source of randomness during key generation to have 2^{15} different seeds

(i.e., 15 bits of entropy).

The RNG is particularly attractive to attackers because it is typically a single isolated hardware or software component easy to locate. The entropy introduced to the program is a critical component of any cryptosystem, and the more the attacker knows about the programs entropy characteristics, the easier attacks become. With the volume of software being produced today, manual effort to identify these flaws (such as what was done to find the Debian OpenSSL bug) cannot scale. It is important to *automatically* identify if a given program applies a certain cryptographic algorithm or uses its random data correctly.

This paper tackles the very first step of this problem by extracting an understanding of how a binary program uses randomness. Specifically, we set the following problem: given a program (or a specific function), estimate bounds on the amount of randomness present in the program or function's output by determining bounds on the *entropy* of this output data. This core technique is meant as a building block upon which the community can start to build program analysis techniques that can begin to reason about the safety of cryptographic functionality, among a number of other applications (for example, estimating the compressibility or error-correction capacity of output data).

Our core contribution is a novel technique to estimate the entropy of the output a single path through a target piece of code, such as a PRNG. Around this, we build a proof-of-concept system process that analyzes the entropy of program output across *many* paths of the program through a multi-step process. First, symbolic execution is used to extract *paths* through target program code. Then, each individual path is analyzed, reasoning not about the values represented by symbolic expressions in the output of the program but about the *entropy* of these values. Finally, the determined entropy of different output paths is combined into a *Multi-path Entropy Distribution*.

The end-to-end system is a minimal proof of concept designed to enable discussion

about our use of symbolic execution to reason about entropy. In its current form, our end-to-end system does not scale to real code and is plagued not only by its own limitations, but also by limitations in the binary analysis engine on which it is implemented. Instead, our aim is to foster conversation about potential uses of symbolic execution in binary analysis beyond crash discovery and testcase generation and to provide an additional building block for future researchers to utilize. To that end, our prototype will be released alongside this paper as a usable extension to the underlying binary analysis engine.

To summarize, this paper makes the following contributions:

- We **design** and introduce a technique to reason about the entropy of a program's output, more precisely than current model-counting and bit-level taint tracking approaches.
- We **implement** a prototype of this technique and evaluate it on several sample programs, one of which approximates the Debian OpenSSL bug.
- We **release**[1] the prototype as open source software to foster advancement in the field.

Before going into the technical details of our technique, we will provide needed background to ensure understanding of our work.

[1] `https://github.com/rhelmot/symtropy`

Chapter 2

Background and Related Work

In this chapter, we describe certain concepts that are necessary for a proper understanding of our technique.

2.1 Randomness and Entropy

The attentive reader will realize that, though our aim is to reason about correct generation and use of random data, we reason about bounds on *entropy*. The relationship between entropy and randomness, and how it impacts cryptographic security, has been studied in the past [2]. While there is not space in this paper to go into detail, the summary is that from an information-theoretical perspective, entropy is necessary, but not sufficient, for randomness.

The implication of this is that a detection of a *low* bound on entropy by our tool almost certainly indicates a problem, but a detection of a *high* bound does not necessarily prove that the code being tested is safe.

Our technique can reason about the amount of entropy in the output of a given chapter of code, but not the amount of randomness. This is because, for the estimation of entropy, it is sufficient to calculate the potential values that can manifest from the output, but to estimate randomness, one must also reason about the *sequential ordering*

of such output[1]. While our technique can reason about the former, we are unaware of any approach that can reason about the latter.

2.2 Program Analysis

Automated program analysis typically falls into two rough categories: *static* and *dynamic* techniques. We briefly describe both categories, with a specific focus on *dynamic symbolic execution*, which is used by our tool.

In this paper, we focus on the analysis of binaries. It is often critical to do security-focused analyses directly on the binary, rather than the corresponding source code, of a target application. It is well established that bugs can be introduced or hidden by the compiler, especially in cryptographic contexts where the use of randomness may register to the compiler as undefined behavior, or where critical operations can be optimized out.

2.2.1 Static Program Analysis

Static analysis reasons about the behavior or safety of large areas of code, usually by interpreting programs over an *abstract domain* [3]. These analysis typically try to deduce *guarantees*: for example, a guarantee that a given piece of code is free of a given type of vulnerability. Static approaches have been proposed to target code issues ranging from integer bounds errors [4] to race conditions [5].

We eschew a static approach because the imprecision of static analysis severely hampers its use in *detecting* issues (as opposed to verifying the absence of issues) [6]. Furthermore, static techniques have difficulty reasoning about programs which are obfuscated or packed, as commercial software tends to be.

[1]Intuitively, a function that output all of its ones before outputting its zeroes would have quite low randomness.

2.2.2 Dynamic Program Analysis

Dynamic analysis is a popular method used for program analysis because it precisely performs security investigation based on run-time information. Two of the most commonly used dynamic analysis techniques are taint analysis and dynamic symbolic execution. Taint analysis performs runtime analysis on the program and determines the effects of predefined tainted input on different computations in a program. Dynamic symbolic execution automatically builds a logical formula describing a program execution path and the relations between the input and the output along that path, which reduces the problem of reasoning about the execution to the domain of logic. The two analyses can be combined to build formulas representing only the parts of an execution that depend upon user-provided values (termed "symbolic values").

The use of dynamic symbolic execution to trace a program processing a concrete input is termed "Concolic Execution", where "Concolic" is a portmanteau of "concrete" and "symbolic". In concolic execution, concrete execution of a program is performed on a specific input and a single control flow path is created. The resulting symbolic values are represented by trees of computation, with concrete values (e.g. `0x10001`) and symbols (e.g. *the third byte of a given input stream*) at their leaves. Execution is performed by a symbolic execution engine, which interprets the target code and maintains for each explored control flow path: (i) a boolean formula that describes the conditions satisfied by the branches taken along that path, and (ii) a symbolic memory store that maps variables to symbolic expressions or values. Branch execution updates the formula, while assignments update the symbolic store.

A dynamic symbolic execution engine can also function independent of a concrete input, in a technique termed "symbolic exploration", which is frequently used for testcase generation. Symbolic execution-based methods for testcase generation gather information

about the execution of a program using a directed search across a state space. Constraint solvers (SMT) are used to check path constraints generated along each execution path, to prune unsatisfiable paths. Even strong SMT solvers such as iSAT and Z3 cannot handle complex, nonlinear constraints efficiently. Since the possible number of execution paths to be considered is so large, only a small part of the program path space can actually be explored (a phenomenon known as "path explosion").

Various techniques have been proposed to mitigate this, from advanced path merging [7] to the approximation of symbolic exploration through the combination of fuzzing and concolic tracing [8]. Any of these path discovery techniques can be used in our tool.

2.3 Model Counting

Our technique attempts to estimate the amount of entropy in the output of a given piece of code. As the measure of entropy is linked to the total number of output values, the clever reader will see the relevance of *model counting*.

Much work has been done on the approximate model-counting problem on SMT formulae, which is effectively equivilent to the problem we are trying to solve. The most recent and most accessible such work is SearchMC [9], a model counting program which works with a number of SMT solvers, and which has an open-source prototype available for evaluation.

2.4 Program Analysis of Cryptography

Program analysis techniques have been previously applied to cryptographic code. For example, a number of static and dynamic techniques to *identify* cryptographic functions and determine their inputs and outputs have been proposed [10, 11, 12, 13, 14, 15]. In an

end-to-end system utilizing our approach, these techniques would be used to identify the code that should be checked against low-entropy input (i.e. a low-entropy source being used for key generation).

Recently, researchers have begun to develop techniques that analyze the proper usage of cryptographic code. One such example, K-Hunt, uses dynamic taint tracking to do a limited check that cryptographic key generation is provided with enough entropy [16]. Since this check is done using dynamic taint tracking, K-Hunt can only reason about the number of bytes used in key generation, and is blind to loss of entropy resulting from complex operations on these keys. In contrast, our technique is able to reason about the amount of entropy provided by *each bit* of input to a cryptographic routine. In K-Hunt's pipeline, our approach would replace their dynamic taint checking pipeline in the determination of, as they term it, "Deterministically Generated Keys".

Finally, techniques have explored other aspects of cryptography-relevant security using program analysis techniques. For example, model counting has been used to quantify the leak of private information in crash reports [17] and to reason about the type of quantity leaked in information disclosure vulnerabilities [18].

Chapter 3

System Overview

In this paper, we describe a proof-of-concept end-to-end system built around our core contribution: the use of symbolic execution, taint tracking, and model counting to estimate bounds on output entropy of a single symbolic path. We detail the end-to-end system here, and cover the specifics of our entropy estimator and the combination of entropy across multiple symbolic paths in future chapters.

Our system expects the following input:

Mandatory: the program. Our prototype is built upon the angr binary analysis framework [6]. This allows us to directly analyze binary code, without need for source code or debug information.

Optional: randomness sources. The places where randomness (and thus, entropy) can be introduced into the program must be provided, so that the system can reason about any reductions in this entropy that occur during program execution. If this is not specified, the system assumes that all input to the program (e.g. bytes read from stdin) represents random input.

Optional: output sinks. To reason about the amount of entropy in the *output* of a piece of code, our system must know what constitutes such output. This could be an instruction address (e.g. a `write` system call), a memory location (e.g. a

destination buffer), or a custom condition. If this is not specified, the system assumes that all output from the program (e.g. bytes written to stdout) represent output sinks.

Give these inputs, our system operates in three stages, as follows. Stages 2 and 3 are further described in their own chapters.

Path discovery using dynamic symbolic execution. First, our system uses symbolic exploration to sample paths from the randomness sources to the output sinks. In our proof-of-concept prototype, this is implemented very naively: angr's default mode of symbolic execution is used with uniform-random concrete constraints on the input sources, producing a representative distribution of paths.

Single-path entropy estimation. We use our core technique of entropy tracking and approximate model counting to estimate the amount of output entropy on every path. This shown in an example in Section 3.1, and described in detail in Chapter 4.

Multi-path entropy estimation. Finally, our system combines the entropy estimations across single paths into an *entropy probability distribution* to reason about the possible amounts of output entropy across multiple program paths. This technique is described in detail in Chapter 5.

As previously mentioned, this end-to-end system is meant to evaluate our core technique on simple sample programs, and does not scale to real software due to a number of reasons, described in Chapter 7.

3.1 Example

Here, we provide a high-level example of the estimation of entropy on a single program path, before describing it in-depth in the next chapter. Throughout this paper we will

use the function `true_rand()` as a function returning a true-random 32 bit integer, each call independent of its previous ones. Consider the following program:

```
int x = true_rand();
x ^= 0x55555555;
x &= 0xffffff;
x |= 0xff0000;
x *= 0x12345678;
write(0, &x, sizeof(x));
```

A casual inspection reveals that there are 16 bits of entropy present in this program's output, even though 32 bits of randomness are generated and there are 32 bits of output. We would like our analysis to be able to tell us this fact.

First, we run through the program with angr's symbolic execution engine. The `true_rand` function is implemented as the creation of a new symbol (`R`), the generation of a random 32 bit number, and the addition of a constraint to the symbolic state that the two be equal. When the emulation finishes, the following AST will be present in the storage region for standard output: `(((R ^ 0x55555555) & 0xffffff) | 0xff0000) * 0x12345678`.

Then, independently of the generated path predicates, we interpret this AST with our abstract operations to recover the bits of information from the input present in the output. The abstraction begins at the root of the tree:

1. The initial variable `R` has 32 bits, and each bit of its abstracted form contains one equality assertion over the corresponding bit of the variable `R`. This will look like a list: `Abstract(R) = [R[0] == 1, R[1] == 1, R[2] == 1, ...]` and so on.

2. The XOR operation preserves the entropy characteristics, so the abstracted form from this operation is the same as the previous one.

3. The implementation of the AND operation knows that `R & 0 == 0`, so it will erase the upper 8 bits from the abstracted form: `Abstract((...) & 0xffffff) = [R[0] == 1, ..., R[23] == 1, 0, 0, 0, ...]`.

4. The implementation of the OR operation knows that `R | 1 == 1`, so it will erase the next 8 bits from the abstracted form: `Abstract((...) | 0xff0000) = [R[0] == 1, ..., R[15] == 1, 1, 1, 1, ..., 0, 0, 0, ...]`

5. the MUL operation scatters bits upwards, since any bit in the multiplication can affect any bits more significant than it in the result: `Abstract((...) * 0x12345678) = [R[0] == 1, R[0] == 1 && R[1] == 1, R[0] == 1 && R[1] == 1 && R[2] == 1, ...]`

The final result indicates that each of the low 16 bits of the output are dependent on between 1 and 16 of the low 16 bits of the initial value of `R`, and high 16 bits of the output are dependent on all of the low 16 bits of the initial value of `R`. There are only 16 bits of assertions present in the 32 bits of the output!

This is a simple example which could also be handled by a basic bit-level dynamic taint tracker. Our technique is designed with the intuition of a bit-level dynamic taint analysis in mind, with improvements in precision and tractability.

Chapter 4

Single-path Entropy Approximation

.

The simplest way to view the problem of counting the amount of entropy present counting the number of possible outputs to a program: the number of bits of entropy present in the program is equivalent to the logarithm (base 2) of the number of possible outputs, if every output is equally likely. This calculation of this number directly is effectively impossible - it is a particularly difficult satisfiability problem known as "model counting".

We can establish some simple upper and lower bounds on this number by rudimentary dynamic analysis. The upper bound is the number of possible inputs, as measured by bits of entropy observed to be used as inputs, for example, by a system-call tracer paired with a dynamic taint tracer (such as what is used in K-Hunt [16]). The lower bound is the number of outputs that can be directly observed by running the program many times. This bound can be probabilistically increased quadratically due to the so-called "birthday paradox", so after n trials and no observed collisions, your best estimate to the number of possible outputs should be n^2. Both of these bounds are fairly *loose*. We focus our efforts on decreasing the upper bound by analyzing the *actual uses* of these bits of entropy via symbolic execution.

4.1 Recovering Information

Our insight is to treat each boolean piece of knowledge from the input present in the output as a bit of entropy. Specifically, our approach is to run a single-path dynamic symbolic execution trace through the program, and then reinterpret the output ASTs using "abstract operations" (implementations of the ordinary SMT operations which produce data in a custom domain, named as such because they are a reduced form of the mechanism underlying Abstract Interpretation) which produce mappings from bits of computer words to the assertions over the input values whose satisfiability uniquely determines the output value. This is key: an ordinary bit-level taint analysis only notes that certain bits are dependent on other bits, but this approach captures the minimal set of assertions that can control the output, a critical insight for reducing the upper bound on an entropy approximation. These assertions are similar in form but fundamentally different from the path predicates generated by dynamic symbolic execution. For ordinary output-bit-is-computed-from-input-bit relations, the assertion is $Extract(input, bit, bit) = 1$. However, if actual boolean expressions appear in the computation (which can happen because of simplifications on the symbolic formula, conditional move instructions, etc), these are also treated as assertions.

An important advantage of the abstract operation approach over a true taint analysis is that it correctly captures loss of information by destructive operations such as OR, AND, MUL, MOD, DIV, etc. The abstract operations understand the ways in which bits are propagated and mixed, and also understand how the presence of certain bits can erase data present in other bits. Abstract operations can also choose to represent the computation as comparisons directly against parts of the AST if they believe that it will produce a more minimal set of assertions. Furthermore, analysis benefits significantly from simplifications that can be performed on the entire formula. This is a huge advantage

of implementing such a taint analysis as a series of abstract operations instead of as a strict dynamic taint analysis.

4.2 Model Counting

Computing the actual quantity of information in the assertions (the entropy) is difficult. The ideal information-theoretic construction is as follows:

- There is a notion of an assertion C
- There is a probability of satisfiability $P(C)$, which is the probability of a random model satisfying the assertion
- There is an information measure $I(C)$, which is the amount of information conveyed by the satisfiability of the assertions (specifically, the fact that the assertions are satisfied), computed from $P(C)$
- From this, the entropy in a series of assertions $H(C_n)$, can be computed from the joint probabilities and information measures of each possible combination of negated and true constraints.

Unfortunately, this is infeasible. First, computing $P(C)$ is itself the model-counting problem, which is currently unsolved in the literature. Second, "each possible combination of negated and true constraints" is a set of 2^n elements. However, the following construction is a reliable upper bound:

- There is a conditional probability measure $P(C_a|C_b)$, the probability of a random model satisfying C_b also satisfying C_a

- There is a conditional information measure $I(C_a|C_b)$, which is the amount of information conveyed by the satisfiability of C_a given that C_b is satisfiable, computed from the joint probability $P(C_a \cap C_b)$ and the probability $P(C_b)$

- There is a conditional entropy measure $H(C_a|C_b)$, which is the amount of uncertainty present in C_a once the satisfiability of C_b is known, computed from the four conditional information measures $I(C_a \cap C_b)$, $I(C_a \cap \neg C_b)$, etc.

- From this, an upper bound on a series of assertions $H_1(C_n)$ can be computed by the sum of the conditional entropies of each assertion conditional on the intersection of all the previous assertions.

- This is necessarily an overapproximation because the predicate in the conditional entropies is necessarily less information than is actually known. The intersection of all the previous assertions is a destructive operation and can be at most a single bit of information.

This has a sum of only n elements! Unfortunately this approach still involves the use of model-counting techniques. While these problems are much smaller than what seem to be typical model-counting workloads, our experiments with the current state of model counting prototypes imply that even this formulation is too complex.

We evaluated a state-of-the-art model counter (specifically, SearchMC [9]), which was unable to give tight bounds on model counting problems that show up often in our output assertions without dozens of minutes of CPU time. Our tests were with simple, single constraints such as $BitVec(A, 32) < 2^{31}$ or $Extract(BitVec(A, 32), 0, 0) == 1$ (both answers should be that there are exactly 2^{31} satisfying models, so $P(C) = 1/2$), and SearchMC produced answers in the billions of solutions.

Instead, we measured the probabilities directly instead of counting models and inferring probability from those. We built a simple stochastic probability tester that simply evaluates the expression for uniform random values of its roots and measures the number of successes. Anecdotally, this approach works well for probabilities that are not very large or very small, and provides acceptable answers for the purpose of evaluating the entropy of an application. Theoretically, this approach takes advantage of an interesting characteristic of this problem — that negligible probabilities have negligible effects on the result.

4.3 Mix Masking

The above approach is good, but it does not solve the problem of *redundant entropy*. Imagine a program with a random number generator seeded as `pseudo_srand(true_rand() ^ true_rand()`. All abstracted outputs will be shown to depend on all 64 bits of entropy that has been generated, even though only the program has only 32 bits of entropy in its output. The program has effectively formed a new entropy source as a reduction of a larger entropy source. To resolve this, whenever more bits of entropy are present in a value than the value is wide, we treat this value as a new independent source of information—a *mix mask*, since it is a mask over several mixed bits of entropy, and since that name sounds very cool.

This has some caveats. Imagine the case of a 512-bit hash being constructed from 32 bits of input, 8 bits at a time. The correct answer is that the output has 32 bits of entropy. Naive mix masking will assign a mask to each of the individual 8 bits of output, and then concatenate them to form an output with 512 different bits of entropy. Our approach is to add the restriction that all mix masks must be independent. If at any point during abstraction we detect that two separate mix masks present in the same value

are rooted in the same bits of input entropy, we incrementally remove the mix masks by replacing their bits with the bits of their original abstractions, until no conflicts remain.

Ideologically, this approach to the mix masking fallback is in conflict with the model-counting approach. It effectively plasters over the nuances of any partial bits of information that fall out of the abstraction with the simple input-bits. The ideal solution to this would be to represent the mix masks not as fresh variables, but rather as the actual AST nodes that were noted to contain too many bits of entropy. These could be fed to the model counter, which could determine their actual information gain over interdependent bits, and no fallback would be needed. However, our entropy approximation is too coarse to handle this correctly. The overapproximation that each new bit adds the information conditional on the intersection of all previous bits works very poorly when we generate many interdependent bits, but very well when we generate a few fresh bits. If we had a better tractable approximation for the entropy of many bits, the algorithm could take a leap forward in precision.

Chapter 5

Multi-path Entropy Estimation

So far, we have only considered analysis of a single path through the program. In this chapter, we extend our technique to combine the estimations of output entropy across multiple paths.

Consider the program in Figure 5.1. How many bits of entropy are present in this program? No answer is sufficient, as this program has a *distribution* of entropy in its output, depending on the number of iterations of the loop executed.

Now consider the program in Figure 5.2. This is an example of a *maliciously* nonrandom program. 1% of the time, it will produce one of 100 (independent) predetermined outputs. This construction creates an asymmetry between the likelihood of usefulness to an attacker in a given output ($1/A$) and the likelihood of a black-box analyst detecting the weakness ($1/(A \times \sqrt{B})$.

Any analysis which wants to reason about cases like these must dip its toes into

Figure 5.1: A program with path-dependent entropy.

```
char val;
do {
        val = true_rand();
        putchar(val);
} while (val);
```

Figure 5.2: A program with malicious entropy reduction.

.

```
int A = 100;
int B = 100;

int (*rand)(void);
int val = true_rand();
if (val % A == 0) {
  pseudo_srand(val / A % B);
  rand = &pseudo_rand;
} else {
  rand = &true_rand;
}
for (int i = 0; i < 100; i++) {
  putchar(rand());
}
```

the world of testcase generation. We note that our problem has different properties than traditional testcase generation: any "interesting" input must actually occur with some non-negligible probability, since entropy cannot be crafted. With this in mind, we randomly sample a large number of inputs uniformly from the input space, so the entropy values in the paths corresponding to each input are a valid sampling of the entire program's entropy.

This is not a sufficient answer in cases where a single path analysis is very slow and a large number of paths must be studied in order to achieve a strong guarantee. Here, one can turn to studied binary analysis and fuzzing techniques for testcase generation. Unless the path predicates can be model-counted in order to determine the probability of each path, this forgoes an actual distribution and can only be used to find the minimum and maximum statistics about the entropy distribution. This is perhaps not a huge problem, since merely the presence of any path with low entropy output could be a sign of an issue.

We do not consider error condition paths, or errors resulting from misconfiguration. If there is any condition of the entropy which will cause the program to output an error, the program should be written in the form of retrying until valid entropy is obtained. This analysis should be considered to run independently on each configuration. In other words: our implementation checks for entropy reduction in an *otherwise well-formed* program, and does not handle other types of errors.

Chapter 6

Evaluation

We evaluted our tool on three basic examples that we believe demonstrate the capabilities and limitations of our technique. Our analysis correctly determines that there are 15 bits of entropy present in the first example (replicating the OpenSSL bug) and that there is the possibility of having a low entropy output in the second and third examples (the programs from chapter 5).

6.1 Case Study: Debian OpenSSL Approximation

Our replication of the OpenSSL bug mentioned in the introduction can be found in Figure 6.1. Every path through the program uses the same amount of entropy, so the entropy sampling returns a flat distribution. This program is, however, a good example for how difficult achieving any semblance of path coverage can be - a branch occurs for each charcter of output, so there are 2^{32} paths.

The actual single-path entropy computation is straightforward — every single bit of the input is present in the output, so the result is just the size of the input symbols that contribute in *any* way to the output — 15 bits.

Interestingly, in this sample, analyzing a single path takes 40 seconds on a modern CPU. The reasons for this are:

Figure 6.1: An approximation of the Debian OpenSSL bug.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

char hexchar(unsigned int val) {
    if (val < 10) {
        return '0' + val;
    } else {
        return 'a' + val - 10;
    }
}

void hexconv(char *out, unsigned char *in, size_t insize) {
    while (insize--) {
        *out++ = hexchar(*in & 0xf);
        *out++ = hexchar(*in++ >> 4);
    }
    *out = 0;
}

int main() {
    srand(getpid());
    unsigned char randbuf[16];
    char outbuf[64];
    int i;

    for (i = 0; i < 16; i++) {
        randbuf[i] = rand();
    }

    hexconv(outbuf, randbuf, 16);
    puts(outbuf);
}
```

1. Because our tool is implemented on top of angr, we are bound by many of the performance limitations inherent in that framework.

2. The constraint checks performed at each program branch (to determine viable paths to explore) spend a significant amount of time in the constraint solver.

3. The stochastic probability evaluator is implemented using replacement operations on angr's Claripy datatypes, which are heavyweight hash-consed ASTs designed for SMT solving.

None of these are performance limitations are inherent to our prototype, and they could potentially be resolved with internal optimizations to angr for single-path execution and concrete evaluation.

6.2 Case Study: Loop-determined Entropy Distribution

Our loop example, in Figure 5.1, with the long-tail entropy distribution, motivates the need for characterizing program entropy as a distribution. Its output is raw, unmutated input, so the abstract operations easily discover the relationships between the input entropy and the program output. Each sampling of the program's entropy will discover a path with a number of loops in it (N), and the amount of entropy in its output will be $8 \times N$ bits. The probability of terminating on any given loop is $\lambda = 1/256$, so the probability density function of N is an $P(n) = \lambda e^{-\lambda x}$.

This does, in fact, match the probability distribution reported by our system.

6.3 Case Study: Malicious Entropy Reduction

Figure 5.2 from Chapter 5 provides a case of *maliciously-injected entropy*. This sample had an issue and had to be tweaked in a subtle way before it could even be analyzed. The definitions of `A` and `B` were originally preprocessor `#define`s, rather than variables. However, this caused the C compiler to optimize the constant division and modulus operations into multiplications by large constants, which the abstract operations were totally unable to reason about in terms of entropy loss. While this specific case could be solved with a optimization pass in angr to detect such division implemenations and replace them with actual division operators, such optimizations are hard to handle in general. In this case, we configured the compiler to insert a real division instruction.

The modulo operation here provides an excellent case study for mix masking, and also an excellent demonstration of its limitations. The correct answer is that the minimum output contains $log_2(100) = 6.64$ bits of entropy. Without mix masking, we detect that the output contains approximately 25 bits of entropy, since the division operation in the seed will be interpreted as causing the lower 25 bits of the output to depend on all of the upper 25 bits of the input, and the modulo operation will be dependent on all of these 25 bits. With mix masking, the modulo operation will be identified as outputing a maximum of 7 bits and masked to a fresh 7 bit value.

Chapter 7

Limitations

Our end-to-end system is intended as a simple proof-of-concept to enable discussion about the core technique of using taint tracking and model counting to reason about novel properties of symbolic expressions. As such, it has a magnitude of limitations, ranging from inapplicability on real code, to performance issues, to insufficencies in our entropy models.

Our intent with this paper is not to provide a turnkey solution for identification of entropy problems in software, but a first step toward the creation of such tools in the future.

7.1 angr-induced Limitations

There are a number of issues introduced by limitations in the underlying binary analysis framework, angr. These are:

- angr's environment model is quite limited, and many system calls are simply unsupported. This causes most programs to be emulated incorrectly.
- angr is impacted by a number of performance issues due to internal implementation details.

- angr is impacted by performance and capability issues of underlying libraries, such as the Z3 constraint solver.

Improvements in angr itself will likewise improve the functionality of our system, but this is outside of the scope of our paper.

7.2 Approximation bounds

The intent of the system is to produce strict over-approximations at all times, but there are some circumstances where it would currently fail to do this, and could be tricked into claiming a lower entropy than actuality. This can currently occur in the abstract implementations of the DIV and MOD operations, and is related to the discussed differences between the assertion propagation analysis and an ordinary taint propagation analysis. This is not necessarily a failure of the approach itself, which only requires that each bit of the output contain the assertions necessary to uniquely determine its value, but instead a failure of the system to implement abstract operations which conform to this requirement.

On the other hand, it would be easy for an attacker to take advantage of the inherant overapproximations of mix-masking in the system. A program with many conditional values of the form `(unsigned int)true_rand() > 0 ? 1 : 0` would appear to our analysis to each contain one bit of entropy, but actually contain 2^{-27} bits of entropy. Addressing this issue requires improving the entropy estimator, as described in Section 4.3. This is a fundamental issue with the approach: there is no upper bound to the over-approximations it produces.

7.3 Inapplicable programs

The above assertion that the analysis should be considered to run on a given program configuration is convenient, but does not allow analysis of programs where output depends on both user input and entropy input. A system capable of reasoning about these scenarios could detect flaws in encryption and signing programs in addition to random number generators.

Chapter 8

Conclusion

Random numbers have obvious importance in cryptographic applications. If a program's PRNG out is predictable, its security may be totally compromised. Yet, such flaws are generally undetectable by current techniques.

In this paper we presented a novel method to analyze the entropy provided by code such as a PRNG using symbolic execution. We have developed a technique to extrapolate the relations between input bits and output bits of a binary by interpreting program output with abstract operations and estimate the entropy of the output via approximate model counting. While existing techniques to analyze cryptographic primitives are focused on identifying program components which perform known cryptographic functions instead of analyzing full-program cryptographic behavior, or are language dependent, our method is free of these limitations.

This work represents an push to apply binary analysis techniques to identify information-theoretical vulnerabilities. The knowledge it recovers, a distribution of output entropy, is useful, but the scope on which the proof-of-concept end-to-end system functions (small programs with fixed user input) is limited. However, generic nature of the approach, requiring only a single-path dynamic symbolic execution pass, makes it highly extensible and can be used as a building block of future work. To that end, we open source the core technique as a library that can be used in conjunction with the angr binary analysis

framework.

Bibliography

[1] S. on Security, "Random number bug in debian linux." `https://www.schneier.com/blog/archives/2008/05/random_number_b.html`.

[2] R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, *Steal this movie: Automatically bypassing drm protection in streaming media services.*, in *USENIX Security Symposium*, pp. 687–702, 2013.

[3] P. Cousot and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252, ACM, 1977.

[4] T. Wang, T. Wei, Z. Lin, and W. Zou, *Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution.*, in *NDSS*, Citeseer, 2009.

[5] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, *Dr. checker: A soundy analysis for linux kernel drivers*, in *26th {USENIX} Security Symposium USENIX Security 17)*, pp. 1007–1024, USENIX Association, 2017.

[6] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, *et. al.*, *Sok:(state of) the art of war: Offensive techniques in binary analysis*, in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 138–157, IEEE, 2016.

[7] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, *Enhancing symbolic execution with veritesting*, in *Proceedings of the 36th International Conference on Software Engineering*, pp. 1083–1094, ACM, 2014.

[8] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, *Driller: Augmenting Fuzzing Through Selective Symbolic Execution*, in *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[9] S. Kim and S. McCamant, *Bit-vector model counting using statistical estimation*, in *Tools and Algorithms for the Construction and Analysis of Systems - 24th*

International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Proceedings (D. Beyer and M. Huisman, eds.), Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pp. 133–151, Springer- Verlag, 1, 2018.

[10] N. Lutz, *Towards revealing attackers intent by automatically decrypting network traffic*, *Master's thesis, ETH Zuerich* (2008).

[11] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, *Reformat: Automatic reverse engineering of encrypted messages*, *Computer Security–ESORICS 2009* (2009) 200–215.

[12] J. Caballero, N. M. Johnson, S. Mccamant, and D. Song, *Binary code extraction and interface identification for security applications*, in *In ISOC NDSS10*, Citeseer, 2010.

[13] J. Caballero, P. Poosankam, S. McCamant, D. Song, *et. al.*, *Input generation via decomposition and re-stitching: Finding bugs in malware*, in *Proceedings of the 17th ACM conference on Computer and communications security*, pp. 413–425, ACM, 2010.

[14] J. Calvet, J. M. Fernandez, and J.-Y. Marion, *Aligot: Cryptographic function identification in obfuscated binary programs*, .

[15] P. Lestringant, F. Guihéry, and P.-A. Fouque, *Automated identification of cryptographic primitives in binary code with data flow graph isomorphism*, in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pp. 203–214, ACM, 2015.

[16] J. Li, Z. Lin, J. Caballero, Y. Zhang, and D. Gu, *K-hunt: Pinpointing insecure cryptographic keys from execution traces*, in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 412–425, ACM, 2018.

[17] M. Castro, M. Costa, and J.-P. Martin, *Better bug reporting with better privacy*, *ACM SIGARCH Computer Architecture News* **36** (2008), no. 1 319–328.

[18] F. Biondi, M. A. Enescu, A. Heuser, A. Legay, K. S. Meel, and J. Quilbeuf, *Scalable approximation of quantitative information flow in programs*, in *International Conference on Verification, Model Checking, and Abstract Interpretation*, pp. 71–93, Springer, 2018.