AQUA: An Automated Tool for Quantifying Leakage in C Programs

Jonathan Heusser

Pasquale Malacaria

{jonathanh,pm}@dcs.qmul.ac.uk

Where we aim to be

```
static int
auth1_process_rhosts_rsa(Authctxt *authctxt, char *info, size_t infolen)
{
        int keybits, authenticated = 0;
        u_int bits;
        Key *client_host_key;
        u_int ulen;

        /*
         * Get client user name.  Note that we just have to
         * trust the client; root on the client machine can
         * claim to be any user.
         */
        client_user = packet_get_string(&ulen);

        /* Get the client host key. */
        client_host_key = key_new(KEY_RSA1);
        bits = packet_get_int();
        packet_get_bignum(client_host_key->rsa->e);
        packet_get_bignum(client_host_key->rsa->n);

        keybits = BN_num_bits(client_host_key->rsa->n);
        if (keybits < 0 || bits != (u_int)keybits) {
                verbose("Warning: keysize mismatch for client_host_key: "
                    "actual %d, announced %d",
                    BN_num_bits(client_host_key->rsa->n), bits);
        }
        packet_check_eom();
```

Secure Programs – Non-Interference

A program is secure iff output observations do not depend on any confidential inputs to the program P.

Such a program is said to be non-interfering.

Joshi and Leino gave a semantic definition of secure information flow for a program P:

```
HH; P; HH = P; HH
```

where `HH` assigns arbitrary value to high h. Thus: only observing the low variables the program should evaluate to the same result no matter what h is assigned to.

Violating Non-Interference → Leakage

Almost every program violates non-interference the question is just by how much?

```
HH; P; HH = P; HH
```

Violation of NI is a distinction you can make on the output for different high inputs.

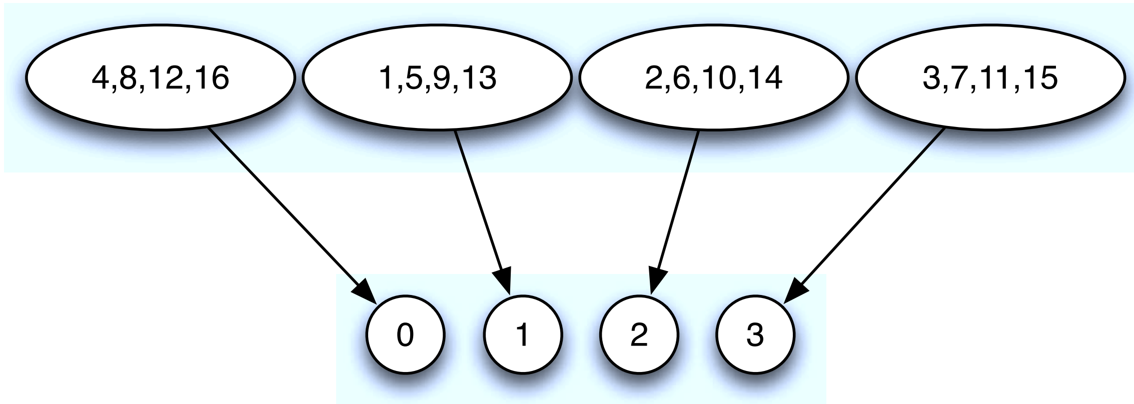
For example if `P(h) = h % 4` then

`P(16) = 0` $\neq$ `P(15) = 3`

Our aim: quantify the amount of violations of NI – i.e. the inference of the input given the outputs

Expanded Example

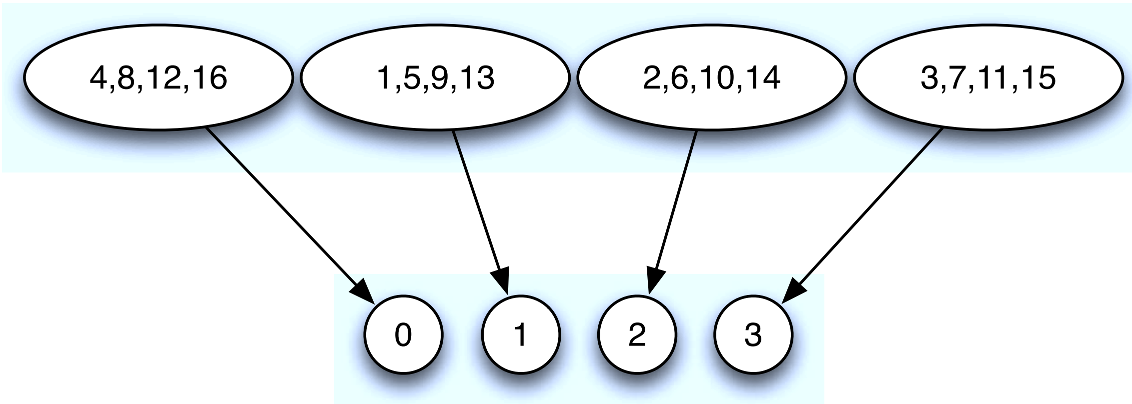
Assume h is 4 bit ($1 \ldots 16$). `P(h) = h % 4`



4 distinctions in the output for 16 input values.
For output 0 what can be learned about the input? It is one of $\{4, 8, 12, 16\}$ out of 16 possible values, i.e. $\frac{1}{4}$.

Expanded Example

Assume h is 4 bit ($1 \ldots 16$). `P(h) = h % 4`



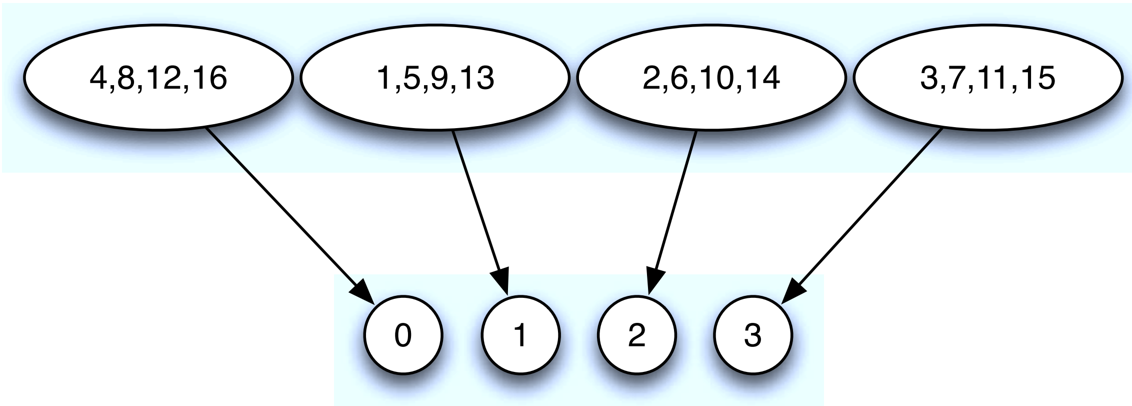
A sensible measure for the information content of an outcome is

$$I(p) = \log_2(\frac{1}{p})$$

Intuitively, the smaller the probability p the larger the information content. In this case: $I(\frac{1}{4}) = \log_2(4) = 2$ bit

Expanded Example

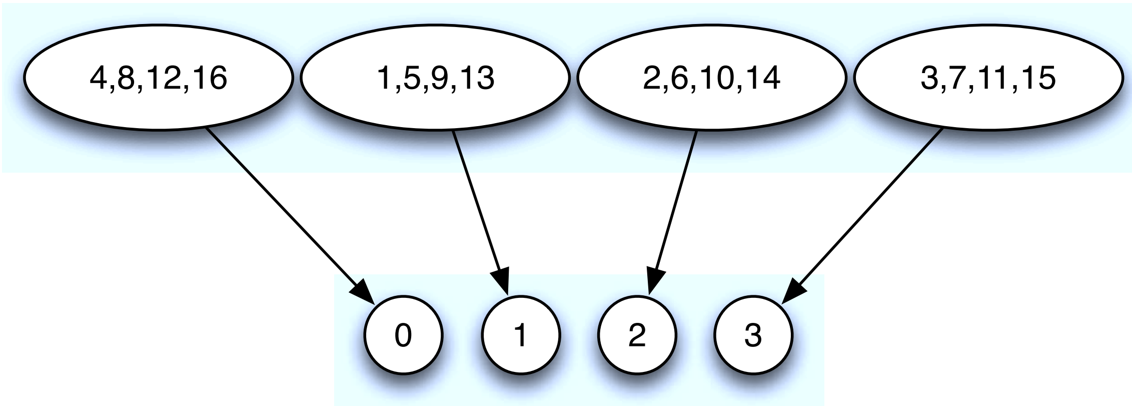
Assume h is 4 bit ($1 \ldots 16$). `P(h) = h % 4`



The information content over all outputs is expressed as expected value $E[I(P)]$

Expanded Example

Assume h is 4 bit ($1 \ldots 16$). `P(h) = h % 4`



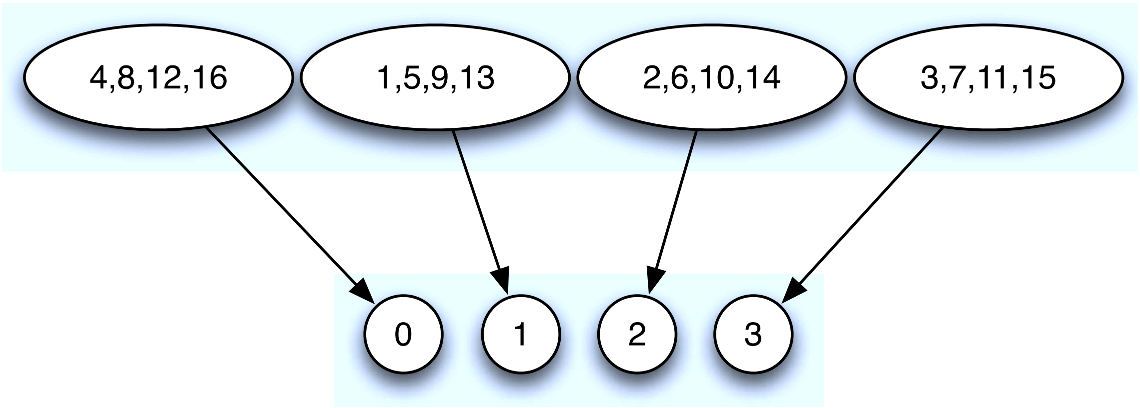
The information content over all outputs is expressed as expected value $E[I(P)]$

$$E[I(P)] = \sum p \; I(\frac{1}{p}) = \sum p \log_2(\frac{1}{p})$$

Weighted information content, what is called Shannon Entropy.

Expanded Example

Assume h is 4 bit ($1 \ldots 16$). `P(h) = h % 4`



The information content over all outputs is expressed as expected value $E[I(P)]$

$$\sum p \log_2(\frac{1}{p}) = 4\ \frac{1}{4} \log_2(4) = 2 \text{ bit}$$

Characterisation of preimage of `P(H)` which *partitions* the high inputs.

Quantifying Leakage and Partitions

Leakage: uncertainty about the inputs after observing the outputs of a program

Measured using Shannon Entropy using the following steps

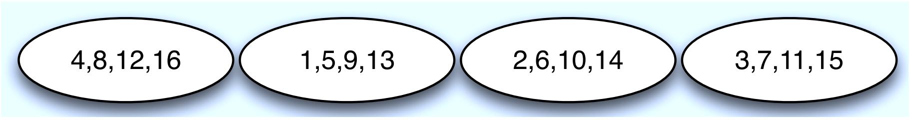
1. Take some code `P(h) = h % 4`

Quantifying Leakage and Partitions

Leakage: uncertainty about the inputs after observing the outputs of a program

Measured using Shannon Entropy using the following steps

1. Take some code `P(h) = h % 4`
2. Find partition on high inputs

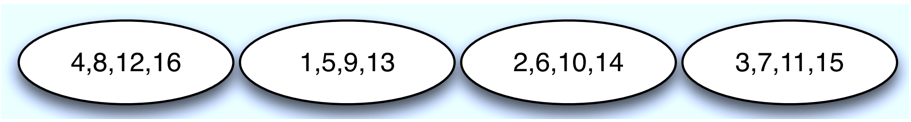


Quantifying Leakage and Partitions

Leakage: uncertainty about the inputs after observing the outputs of a program

Measured using Shannon Entropy using the following steps

1. Take some code `P(h) = h % 4`
2. Find partition on high inputs



3. Quantify using Entropy

$$\sum p \log_2(\frac{1}{p})$$

From Programs to Partitions

Given a partition and input probability distribution, quantification is simple. Just plug-in your measure.

More difficult is to get the partition for a program:

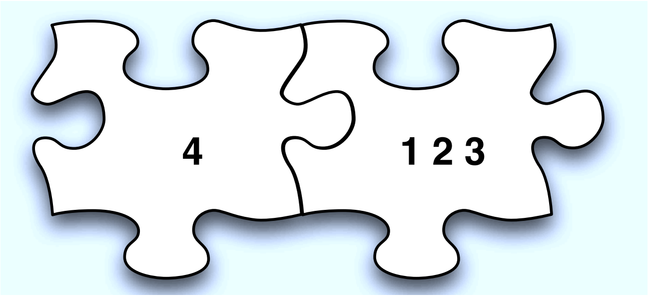
$$\Pi : Program \rightarrow Partition$$

Tool to calculate $\Pi(P)$ for subset of ANSI-C programs.

Automatically Calculating $\Pi(P)$

With 2 bit `pin`,

$$P \equiv$$ `if(pin==4) ok else ko`

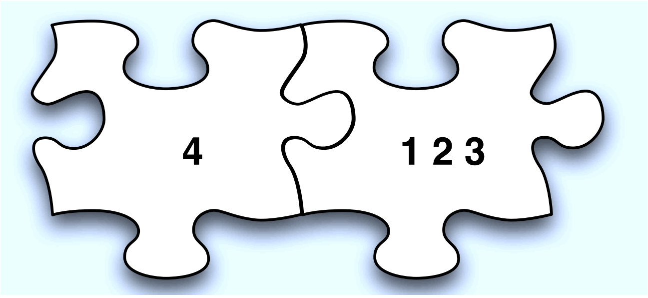


Partition defined by *number* and *sizes* of equivalence classes

Automatically Calculating $\Pi(P)$

With 2 bit `pin`,

$$P \equiv \texttt{if(pin==4) ok else ko}$$



Partition defined by *number* and *sizes* of equivalence classes

Two step approach:

- Find a representative input for each possible output
- For each found input, count how many other inputs lead to the same output

Self-Composition and Reachability

The original NI check:

```
HH; P; HH = P; HH
```

Program P appears twice, i.e. NI violation is detected by observing two execution paths in P.

Self-Composition and Reachability

The original NI check:

```
HH; P; HH = P; HH
```

Program P appears twice, i.e. NI violation is detected by observing two execution paths in P.

NI definition is a 2-safety property: can be refuted observing two finite program runs, can be checked using reachability analysis

h=α; h'=β; P; P'; if(l $\neq$ l') $\{$ NI_ERROR $\}$

Approach called *self-composition* by Barthe et al.

Automatically Calculating $\Pi(P)$

Create two instances $P_{\neq}$ and $P_{=}$ out of P applying self-composition, inputs are h, h' and ouputs l, l'

$$P_{\neq}(i) \equiv h = i; P; P'; \texttt{assert}(\texttt{l} \neq \texttt{l}')$$
$$P_{=}(i) \equiv h = i; P; P'; \texttt{assert}(\texttt{l} = \texttt{l}')$$

translated to SAT queries for SAT solving and model counting.

$P_{\neq}$ responsible for finding set of representative inputs S_{input} with unique outputs ($l \neq l'$)

$P_{=}$ model counts every element of S_{input}

Algorithm for $P_{\neq}$ by example

$P \equiv$ `if(h==4) 0 else 1`

Input: $P_{\neq}$
Output: S_{input}
$S_{input} \leftarrow \emptyset$
$h \leftarrow random$
$S_{input} \leftarrow S_{input} \cup \{h\}$
while $P_{\neq}(h)$ *not unsat* **do**
    $(l, h') \leftarrow$ Run SAT solver on $P_{\neq}(h)$
    $S_{input} \leftarrow S_{input} \cup \{h'\}$
    $h \leftarrow h'$
    $P_{\neq} \leftarrow P_{\neq} \wedge l' \neq l$
end

$S_{input} = \{0, 4\}$ thus P has two equivalence classes

S_{input} is input to the algorithm for $P_{=}$

Algorithm for $P_=$ by example

$$P \equiv \texttt{if(h==4) 0 else 1}$$

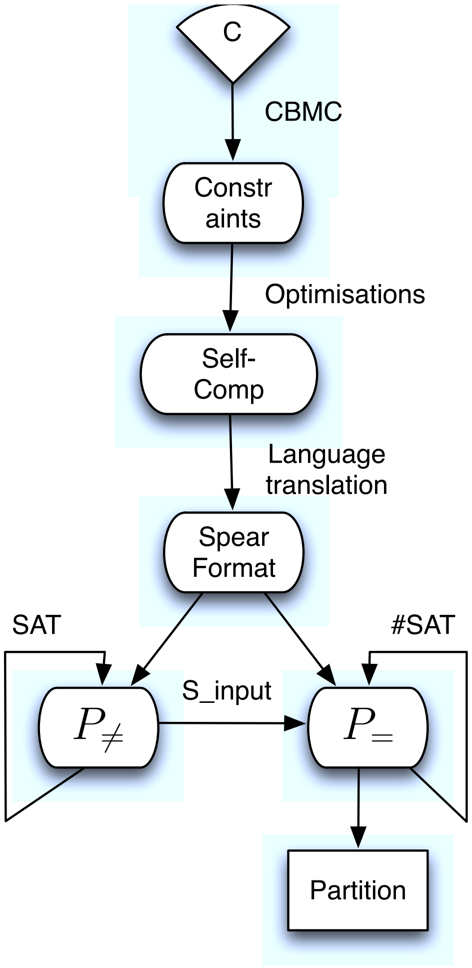
$$S_{input} = \{0, 4\}$$

Input: $P_=$, S_{input}
Output: M
$M = \emptyset$
while $S_{input} \neq \emptyset$ **do**
  $h \leftarrow s \in S_{input}$
  $\#models \leftarrow$ Run $allSAT$ solver on $P_=(h)$
  $M = M \cup \{\#models\}$
  $S_{input} \leftarrow S_{input} \setminus \{s\}$
end

Partition for program P is $M = \{$1 model$\}\{$3 models$\}$

Demo

Implementation: AQUA



Main features & constraints

- runs on subset of ANSI-C, without memory alloc, only integer secrets, no interactive input
- no annotations needed except cmdline options
- supports non-linear arithmetic and integer overflows
- Tool chain: CBMC, Spear, RelSat, C2D
- Computation easily distributed

Loops and Soundness

Bounded loop unrolling is a source of unsoundness: not all possible behaviours are considered.

```
l=0; while(l < h) { l++; }
      ⇓
l=0; if(l < h) { l++; if(l < h) { l++; ...
```

All untreated inputs end up in a "sink state".
Program above with 4 bit variables and 2 unrollings generates partition: $\{1\}\{1\}\color{red}{\{14\}}$

Entropy can be over-approximated by distributing the sink state into singletons: $\{1\}\{1\}\underbrace{\{1\}\ldots\{1\}}_{14\text{x}}$

From C to SPEAR

```
int main() {
    int h1,h2,h3,l;
    l = h1+h2+h3;
}
```

CBMC translates C to SSA constraints

```
tmp11 == (h110 + h210)
l11 == (h310 + tmp11)
```

For loops are unrolled completely, *while loops* up to user defined iteration.
CBMC is *not* used for model checking here!

Generate $P_{\neq}$ by translating intermediate language above

$P_{\neq}$ in SPEAR Format

```
d l11__:i12 tmp11__:i12 l11:i12 tmp11:i12 ...
p = h310 0:i12 # secret initialisations
p = h210 0:i12
p = h110 0:i12
p ule h310 5:i12 # constraining domain
p ule h310__ 5:i12
..
c tmp11 + h110 h210 # self composed program
c l11 + h310 tmp11
c tmp11__ + h110__ h210__
c l11__ + h310__ tmp11__
p /= l11__ l11
```

$P_{\neq}$ in SPEAR Format

```
d l11__:i12 tmp11__:i12 l11:i12 tmp11:i12 ...
p = h310 0:i12 # secret initialisations
p = h210 0:i12
p = h110 0:i12
p ule h310 5:i12 # constraining domain
p ule h310__ 5:i12
..
c tmp11 + h110 h210 # self composed program
c l11 + h310 tmp11
c tmp11__ + h110__ h210__
c l11__ + h310__ tmp11__
p /= l11__ l11
```

model found:

```
h110__=5, h210__=5, h310__=5, l11__=15
```

$P_{\neq}$ in SPEAR Format

```
d l11__:i12 tmp11__:i12 l11:i12 tmp11:i12 ...
p = h310 5:i12 # secret initialisations
p = h210 5:i12
p = h110 5:i12
p ule h310 5:i12 # constraining domain
p ule h310__ 5:i12
..
c tmp11 + h110 h210 # self composed program
c l11 + h310 tmp11
c tmp11__ + h110__ h210__
c l11__ + h310__ tmp11__
p /= l11__ l11
```

blocking clauses to not find same solutions again

```
p /= l11__ 15:i12
```

$P_{=}$ in SPEAR Format

```
d l11__:i12 tmp11__:i12 l11:i12 tmp11:i12 ...
p = h310 ?:i12 # secret initialisations
p = h210 ?:i12
p = h110 ?:i12
p ule h310 5:i12 # constraining domain
p ule h310__ 5:i12
..
c tmp11 + h110 h210 # self composed program
c l11 + h310 tmp11
c tmp11__ + h110__ h210__
c l11__ + h310__ tmp11__
p = l11__ l11
```

translated to CNF and fed to model counters (relsat, c2d)

Estimating Entropy

Complete enumeration via $P_{\neq}$ is not needed to calculate the entropy approximatively

Idea: only "sample" n equivalence classes through $P_{\neq}$. Use the partial representation of partition to estimate entropy of the whole secret space.

Normal sampling: $\{\ldots 1 \ldots\}\{\ldots 1 \ldots\}\{\ldots 2 \ldots\}\cdots$

Sampling equivalence classes: $\{5\}\{5\}\{6\}$

Estimating Entropy

Example: Sample S with 3 equivalence classes to get the partition on an input space of 7 bit (128 unique inputs).

$$\{5\}\{5\}\{6\} \qquad (\frac{5}{128}, \frac{5}{128}, \frac{6}{128})$$

Intuition: Estimate remaining number of equivalence classes proportional to the sample S and distribute remaining inputs equally.

Estimating Entropy

Example: Sample S with 3 equivalence classes to get the partition on an input space of 7 bit (128 unique inputs).

$$\{5\}\{5\}\{6\} \qquad (\frac{5}{128}, \frac{5}{128}, \frac{6}{128})$$

Intuition: Estimate remaining number of equivalence classes proportional to the sample S and distribute remaining inputs equally.

3 eq. classes sampled with coverage $\frac{5+5+6}{128} = \frac{1}{8}$
Remaining $\frac{7}{8}$ of inputs (112) will be split in $7 * 3 = 21$ equivalence classes $\rightarrow$ CRC8 demo.

Conclusions

- Automated tool built on SAT solving and model counting to calculate entropy
- Entropy estimators can improve performance significantly for certain programs