Leakage in presence of an active and adaptive adversary

PACS Team - Verimag Laboratory
e-mail : Cristian.Ene@univ-grenoble-alpes.fr
Laurent.Mounier@univ-grenoble-alpes.fr

October 6, 2021

Measuring the information leakage of a system is very important for security. From side-channels to biases in random number generators, quantifying how much information a system leaks about its secret inputs is crucial for preventing adversaries from exploiting it; this has been the focus of intensive research efforts in the areas of privacy and of quantitative information flow (QIF). For example, both programs in Figure 1 are leaking some additional information about the secret if one can measure the execution time or if one can observe the instruction cache. Moreover, by interacting iteratively with the application, the adversary is able to improve his knowledge [6].

```
void compare(int l, int  s){
  if (s<l)
      {write_log(``too large'');}    // 1 sec.
  else
      {some_computation();}          // 2 sec.
}
```

```
int pwdCheck(char *l, char*  pwd){
  unsigned i;
  for (i=0; i<B_Size; i++)
       if (l[i]!=pwd[i])
            {return 0;}
  return 1;
}
```

Figure 1: Leaking programs

Hence the overall scenario (Figure 2) is the following one:

- Some secret $x \in \mathcal{X}$ is generated and provided to the application
- Iteratively and adaptively,
  1. The adversary provides some public input $l \in \mathcal{L}$ to the application
  2. The application does some computation and outputs some $y \in \mathcal{Y}$

The adversary's knowledge about the the secret $x \in \mathcal{X}$ at some moment i is called the prior probability π_i (e.g. initially, π_0 would be the uniform distribution on $\mathcal{X}$). In our context, the application corresponds to a family of probabilistic channels $(\mathcal{C}_l)_{l\in\mathcal{L}}$, such that for each $x \in \mathcal{X}$ and $l \in \mathcal{L}$, it returns a $y \in \mathcal{Y}$ according to some distribution $\mathcal{P}_{\mathcal{C}_l}(Y = y \mid X = x)$. In the considered

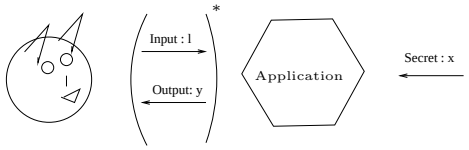

Figure 2: The target scenario

scenario, the adversary interacts iteratively (Figure 3) with the application until his knowledge π_k achieves some desired vulnerability level $\mathcal{V}(\pi_k)$.

```
π←π0;                                              // (1)
while V(π) ≤ ε do                                  // (2)
    l0←argmax_{l∈L}  V[π▷Cl];                      // (3)
    Execute App with input l0;
    Get the output y0 returned by App;
    Update π according to  y0                      // (4)
```

where

- (1) π_0 is the initial probabilistic information about the secret $x[1]$ (called the prior)
- (2) ϵ is the intended level of knowledge (modelled by some measure $\mathcal{V}$) about the secret
- (3) find the "best" input l_0 that optimises the leakage ; $\pi \triangleright \mathcal{C}_{l_0}$ is the hyper-distribution corresponding to executing App with prior π and input l_0, i.e. the distribution of posteriors $\mathcal{P}(X \mid Y = y_0, L = l_0)$, each with probability $\mathcal{P}(Y = y_0 \mid L = l_0)$
- (4) use the Bayes law to update the belief: $\pi \leftarrow \mathcal{P}(X \mid Y = y_0, L = l_0)$

Figure 3: Attacker's strategy

Several issues can be investigated in this internship:

- What are the best choices for the measures $\mathcal{V}$ and $\mathcal{V}[\pi \triangleright C]$?
- How to compute/approximate for each input l, the associated probabilistic channel $\mathcal{C}_l$ (do we have the source code or not for App)?
- Given that in most of the cases the sets of inputs $\mathcal{L}$, secrets $\mathcal{X}$ and observables $\mathcal{Y}$ can be very large
  - How to represent and manipulate the distributions/hyper-distributions over $\mathcal{L}$?
  - How to compute/approximate $argmax_{l \in \mathcal{L}}\ \mathcal{V}[\pi \triangleright \mathcal{C}_l]$, knowing that in the most realistic scenario, the output Y will rather be a continuous random variable [2].

- Information-theoretic vs. probabilistic polynomial-time adversary.

The topic of this internship can be oriented in various directions:

- refine the scenario from Figure 3 in a grey-box case, where the attacker has the binary code of the application and integrate it into the BinSec platform [3].
- develop abstractions and tools to represent and manipulate distributions over memories where variables take values over large domains [5, 8].
- implementing the scenario from Figure 3 in order to synthesis an adaptive attack [7] or to measure the vulnerability for a concrete application, using for example symbolic execution [4].

This internship can be done within the framework of project ANR 20-CE25-0009 TAVA.

References

[1] Mário S. Alvim et al. "An Axiomatization of Information Flow Measures". In: *Theoretical Computer Science* 777 (2019), pp. 32–54. DOI: 10.1016/j.tcs.2018.10.016. URL: https://hal.archives-ouvertes.fr/hal-01995712.

[2] Lucas Bang, Nicolás Rosner, and Tevfik Bultan. "Online synthesis of adaptive side-channel attacks based on noisy observations". In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2018, pp. 307–322.

[3] Robin David et al. "BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE. 2016, pp. 653–656.

[4] James C King. "Symbolic execution and program testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[5] Piotr Mardziel et al. "Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation". In: *Journal of Computer Security* 21.4 (2013), pp. 463–532.

[6] Quoc-Sang Phan et al. "Synthesis of adaptive side-channel attacks". In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE. 2017, pp. 328–342.

[7] Seemanta Saha et al. "Incremental Attack Synthesis". In: *ACM SIGSOFT Software Engineering Notes* 44.4 (2019), pp. 16–16.

[8] Ian Sweet et al. "What's the over/under? probabilistic bounds on information leakage". In: *International Conference on Principles of Security and Trust*. Springer, Cham. 2018, pp. 3–27.