Quantifying Information Flow for Dynamic Secrets

Piotr Mardziel,[†] Mário S. Alvim,[‡] Michael Hicks,[†] and Michael R. Clarkson[*]
[†]University of Maryland, College Park
[‡]Universidade Federal de Minas Gerais
[*]George Washington University

***Abstract*—A metric is proposed for quantifying leakage of information about secrets and about how secrets change over time. The metric is used with a model of information flow for probabilistic, interactive systems with adaptive adversaries. The model and metric are implemented in a probabilistic programming language and used to analyze several examples. The analysis demonstrates that adaptivity increases information flow.**

***Keywords*—dynamic secret, quantitative information flow, probabilistic programming, gain function, vulnerability**

I. Introduction

Quantitative information-flow models [1]–[5] and analyses [6]–[9] typically assume that secret information is *static*. But real-world secrets evolve over time. Passwords, for example, should be changed periodically. Cryptographic keys have periods after which they must be retired. Memory offsets in address space randomization techniques are periodically regenerated. Medical diagnoses evolve, military convoys move, and mobile phones travel with their owners. Leaking the current value of these secrets is undesirable. But if information leaks about how these secrets change, adversaries might also be able to predict future secrets or infer past secrets. For example, an adversary who learns how people choose their passwords might have an advantage in guessing future passwords. Similarly, an adversary who learns a trajectory can infer future locations. So it is not just the current value of a secret that matters, but also how the secret changes. Methods for quantifying leakage and protecting secrets should, therefore, account for these *dynamics*.

This work initiates the study of quantitative information flow (henceforth, QIF) for dynamic secrets. First, we present a core model of programs that compute with dynamic secrets. We use *probabilistic automata* [10] to model program execution. These automata are interactive: they accept inputs and produce outputs throughout execution. The output they produce is a random function of the inputs. To capture the dynamics of secrets, our model uses *strategy functions* [11] to generate new inputs based on the history of inputs and outputs. For example, a strategy function might yield the GPS coordinates of a high-security user as a function of time, and of the path the user has taken so far.[1]

Our model includes *wait-adaptive adversaries*, which are adversaries that can observe execution of a system, waiting until a point in time at which it appears profitable to attack. For example, an attacker might delay attacking until collecting enough observations of a GPS location to reach a high confidence level about that location. Or an attacker might passively observe application outputs to determine memory layout, and once determined, inject shell code that accesses some secret.

Second, we propose an information-theoretic metric for quantifying flow of dynamic secrets. Our metric can be used to quantify leakage of the current value of the secret, of a secret at a particular point in time, of the history of secrets, or even of the strategy function that produces the secrets. We show how to construct an optimal wait-adaptive adversary with respect to the metric, and how to quantify that adversary's expected *gain*, as determined by a scenario-specific *gain function* [14]. These functions consider when, as a result of an attack, the adversary might learn all, some, or no information about dynamic secrets. We show that our metric generalizes previous metrics for quantifying leakage of static secrets, including vulnerability [4], guessing entropy [15], and g-vulnerability [14]. We also show how to limit the power of the adversary, such that it cannot influence inputs or delay attacks.

Finally, we put our model and metric to use by implementing them in a probabilistic programming language [16] and conducting a series of experiments. Several conclusions can be drawn from these experiments:

- Frequent change of a secret can increase leakage, even though intuition might initially suggest that frequent changes should decrease it. The increase occurs when there is an underlying order that can be inferred and used to guess future (or past) secrets.
- Wait-adaptive adversaries can derive significantly more gain than adversaries who cannot adaptively choose when to attack. So ignoring the adversary's adaptivity (as in prior work on static secrets) might lead one to conclude secrets are safe when they really are not.
- A wait-adaptive adversary's expected gain increases monotonically with time, whereas a non-adaptive adversary's gain might not.
- Adversaries that are *low adaptive*, meaning they are capable of influencing their observations by providing low-security inputs, can learn exponentially more information than adversaries who cannot provide inputs.

We proceed as follows. Section II reviews QIF for static secrets and motivates the improvements we propose. Section III

[1]Our probabilistic model of interaction is a refinement of the nondeterministic model of Clark and Hunt [12], and it is a generalization of the interaction model of O'Neill et al. [13]. See Section VII for details.

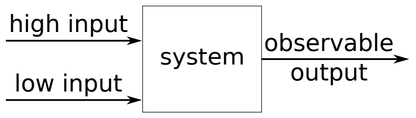
presents our model of dynamic secrets, and Section IV presents our metric for leakage. Section V describes our implementation and Section VI presents our experimental results. Section VII discusses related work, and Section VIII concludes.

II. Quantitative Information Flow

Consider a password checker, which grants or forbids access to a user based on whether the password supplied by the user matches the password stored by the system. The password checker must leak secret information, because it must reveal whether the supplied password is correct. Quantifying the amount of information leaked is useful for understanding the security of the password checker—and of other systems that, by design or by technological constraints, must leak information.

A. QIF for static secrets

The classic model for QIF, pioneered by Denning [17], represents a system as an information-theoretic channel:



A *channel* is a probabilistic function. The system is a channel, because it probabilistically maps a high security (i.e., secret) input and a low security (i.e., public) input to an observable (i.e., public) output. (High security outputs can also be modeled, but we have no need for them in this work.) The adversary is assumed to know this probabilistic function.

The adversary is assumed to have some initial uncertainty about the high input. By providing low input and observing output, the adversary derives some revised uncertainty about the high input. The change in the adversary's uncertainty is the amount of leakage:

$$\textit{leakage} = \textit{initial uncertainty} - \textit{revised uncertainty}.$$

Uncertainty is typically represented with probability distributions [18]. Specific metrics for QIF use these distributions to calculate a numeric quantity of leakage [1]–[5].

More formally, let X_{H}, X_{L} and X_{O} be random variables representing the distribution of high inputs, low inputs, and observables, respectively. Given a function $F(X)$ of the uncertainty of X, leakage is calculated as follows:

$$F(X_{\mathrm{H}}) - F(X_{\mathrm{H}} \mid X_{\mathrm{L}} = \ell, X_{\mathrm{O}}), \qquad (1)$$

where ℓ is the low input chosen by the adversary, $F(X_{\mathrm{H}})$ is the adversary's initial uncertainty about the high input, and $F(X_{\mathrm{H}} \mid X_{\mathrm{L}} = \ell, X_{\mathrm{O}})$ is the revised uncertainty. As is standard, $F(X_{\mathrm{H}} \mid X_{\mathrm{L}} = \ell, X_{\mathrm{O}})$ is defined to be $\mathbb{E}_{o \leftarrow X_{\mathrm{O}}}[F(X_{\mathrm{H}} \mid X_{\mathrm{O}} = o, X_{\mathrm{L}} = \ell)]$, where $\mathbb{E}_{x \leftarrow X}[f(x)]$ denotes $\sum_x \Pr(X = x) \cdot f(x)$.

Various instantiations of F have been proposed, including Shannon entropy [1]–[3], [19]–[22], guessing entropy [23], [24], marginal guesswork [25], vulnerability [4], [26], and g-leakage [14].

B. Toward QIF for dynamic secrets

There are several ways in which the classic model for QIF is insufficient for reasoning about dynamic secrets:

- **Interactivity:** Since secrets can change, the adversary should be able to choose inputs based on past observations. That is, we want to allow *feedback* from outputs to inputs. The classic model doesn't permit feedback. There are some QIF models for interactivity; we discuss them in Section VII.
- **Input vs. attack:** Classic QIF metrics quantify leakage with respect to a single low input from the adversary. Each input is an *attack* made by the adversary to learn information. But with an interactive system, some low inputs might not be attacks. For example, an adversary might navigate through a website before uploading a maliciously crafted string to launch a SQL injection attack; the navigation inputs themselves are not attacks. Our model naturally supports quantification of leakage at the times when attacks occur.
- **Delayed attack:** Combining the above two features, adversaries should be permitted to adaptively choose when to attack based on their interaction with the system, and this decision process should be considered when quantifying leakage.
- **Moving target:** New secrets potentially replace old secrets. The classic model cannot handle these *moving targets.* To quantify leakage about moving-target secrets, we need a model of how secrets evolve over time. Prior work [27], [28] has considered leakage only about the entire stream of secrets, rather than a particular value of a secret at a particular time.

To address these insufficiencies in the classic model, we introduce a new model for QIF of dynamic secrets.

III. Model of Dynamic Secrets

Our model of dynamic secrets involves a *system*, which executes within a *context*. The system represents a program, such as a password checker. The context represents the program's environment, which includes agents such as users and adversaries. During an *execution*, the system and context interact to produce traces of inputs and outputs. The system receives inputs from the context, and it yields outputs back to that context. With the password checker, for example, a password file might be a source of high inputs, and the adversary might be a source of low inputs—specifically, of guesses in an online guessing attack.

A. Systems as fully probabilistic automata

As in the classic model of QIF, a system accepts a high input and a low input, and probabilistically produces an observable output. Let $\mathcal{H}$, $\mathcal{L}$, and $\mathcal{O}$ be finite sets of high inputs, low inputs, and observable outputs.

Our model adds a notion of logical time to the classic model. At each time step t of an execution, three *events* occur sequentially: (i) the system accepts a high input h_t; (ii) the system accepts a low input ℓ_t; and (iii) the system produces

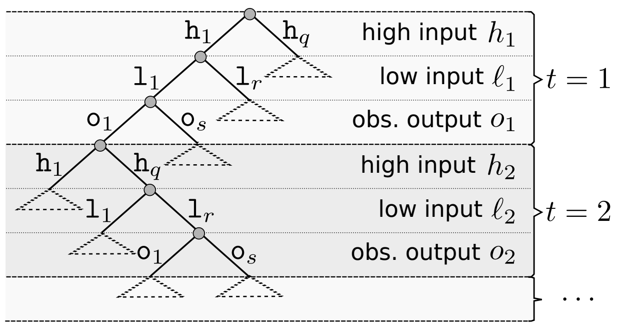
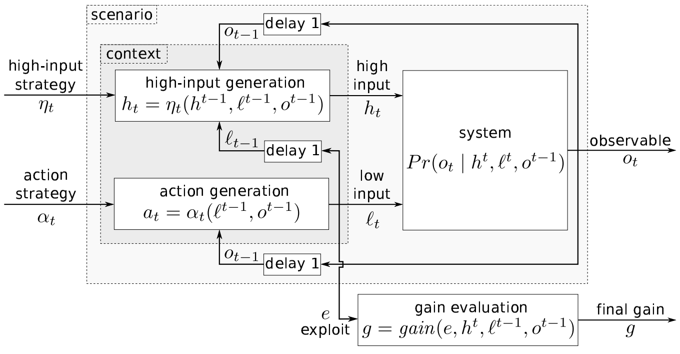


Fig. 1. A tree depicting all possible executions of a fully probabilistic automaton. Each edge would also be labeled with a probability, which is omitted from the figure for simplicity.



scenario
context
delay 1
o_{t-1}
high-input strategy η_t
high-input generation $h_t = \eta_t(h^{t-1}, \ell^{t-1}, o^{t-1})$
high input h_t
system $Pr(o_t \mid h^t, \ell^t, o^{t-1})$
observable o_t
ℓ_{t-1}
delay 1
action strategy α_t
action generation $a_t = \alpha_t(\ell^{t-1}, o^{t-1})$
low input ℓ_t
o_{t-1}
delay 1
e exploit
gain evaluation $g = gain(e, h^t, \ell^{t-1}, o^{t-1})$
final gain g

Fig. 2. Model of dynamic secrets. The arrows feeding high inputs, low inputs, and observables to the gain function are omitted for simplicity.

an observable output o_t. An execution lasting T time steps thus produces an execution *history* (synonomously, a *trace*) of the following form:

$$\underbrace{h_1\ \ell_1\ o_1}_{t=1}\ \underbrace{h_2\ \ell_2\ o_2}_{t=2}\ \ldots\ \underbrace{h_T\ \ell_T\ o_T}_{t=T}.$$

Our assumption of cyclic alternation between high inputs, low inputs and observable outputs does not preclude executions where these events happen in other orders. To model such executions, it suffices to define dummy inputs or outputs that are used whenever an event is skipped.

The execution of a system within a context can be represented using a *fully probabilistic automaton* [10].[2] The execution of such an automaton can be represented as a tree where any path from the root to a leaf corresponds to an execution history, as depicted in Figure 1. Each edge in the tree corresponds to an event in the history. Let x^t denote the sequence of events of type x up to time t—for example, $h^t = h_1, \ldots, h_t$. Denote the probability of an event as follows:

- **High inputs:** $\Pr(h_t \mid h^{t-1}, \ell^{t-1}, o^{t-1})$ is the probability of high input h_t occurring, conditioned on the execution history $(h^{t-1}, \ell^{t-1}, o^{t-1})$ through time $t-1$.
- **Low inputs:** $\Pr(\ell_t \mid \ell^{t-1}, o^{t-1})$ is the probability of low input ℓ_t occurring, conditioned on the public execution history (ℓ^{t-1}, o^{t-1}) through time $t-1$. Since this probability is not conditioned on h^{t-1}, low inputs cannot depend on past high inputs.
- **Observable outputs:** $\Pr(o_t \mid h^t, \ell^t, o^{t-1})$ is the probability of the system producing observable output o_t, conditioned on the execution history $(h^{t-1}, \ell^{t-1}, o^{t-1})$ up to time $t-1$, as well as the high input h_t and low input ℓ_t occurring at time t.

Given the probabilities of events, the probability of an execution (h^T, ℓ^T, o^T) is defined as follows:

$$\Pr(h^T, \ell^T, o^T) = \prod_{t=1}^{T} [\Pr(h_t \mid h^{t-1}, \ell^{t-1}, o^{t-1}) \cdot \Pr(\ell_t \mid \ell^{t-1}, o^{t-1}) \cdot \Pr(\ell_t \mid h^t, \ell^t, o^{t-1})]. \tag{2}$$

B. Interaction of a system with the environment

An execution context comprises a pair of functions—the *high-input strategy* [11] and the *action strategy*—that generate inputs for the system. These functions represent the high and low agents in the environment. Our interaction model makes these functions explicit, whereas they are left implicit in a fully probabilistic automaton.[3] Figure 2 depicts the interaction of a system with its context. The high-input strategy η_t produces a high input h_t at each time step t as a function of the history $(h^{t-1}, \ell^{t-1}, o^{t-1})$ of execution. Notice that the strategy can change at each time step; we return to this point, below.

The action strategy models the distinction between attacks and inputs. At each time step, the action strategy produces a new input on behalf of the adversary, resulting in a new observable from the system. That observable is fed back into the action strategy at the next time step. Eventually, the adversary has gathered enough observations and commits to an attack, represented by the strategy producing an *exploit*. (We leave modeling interleaved attacks and low inputs as future work.) Let $\mathcal{E}$ be set of exploits. Define an *action* to be either a low input or an exploit, and let $\mathcal{A}$ be the set of actions, where $\mathcal{A} = \mathcal{L} \cup \mathcal{E}$. At each time step, the action strategy α_t produces an action a as a function of the public history (ℓ^{t-1}, o^{t-1}) of execution. As with high-input strategies, there can be a different action strategy at each time step.

The adversary in our model is wait adaptive: it can pick the best time to attack. An adversary that is not wait adaptive, as in the classic model of QIF, can attack only at a fixed time. Similarly, the adversary in our model is *low adaptive*: it can generate low inputs, based on past observations, that influence the system. An adversary that is not low adaptive could only passively observe the system prior to attack.

[2]The execution of a system within an unknown context can be modeled by an automaton in which the events corresponding to the production of inputs are nondeterministic hence are not labeled with probabilities. Such automata can be used to reason about the maximum leakage of a system over all possible contexts [28].

[3]In the accompanying technical report [29] we show how to extract the context from a fully probabilistic automaton.

Our model employs a time-indexed sequence of deterministic strategy functions. Such a sequence can be used to emulate a single probabilistic strategy by assuming the sequence is generated by making a probabilistic choice, at each time step, from among a set of deterministic strategies. Indeed, the equivalence between the two approaches is stated by Theorem 2 in Section III-E. Deterministic strategies are more convenient for proving mathematical properties of the model, so we use them in the remainder of this section. Probabilistic strategies, on the other hand, are more convenient for expressing our metrics, and also make it more straightforward to reason about increasing knowledge (i.e., decreased uncertainty) of the strategy and the secrets it generates. Section IV and the following sections use probabilistic strategies.

C. Success of the adversary

Once the adversary commits to an exploit e, no more observations take place. The success of the adversary is now determined. Define a *scenario* to be the history $(h^t, \ell^{t-1}, o^{t-1})$ of execution up to the exploit. The number of high inputs in a scenario is one more than the number of low inputs and number of observations, because at the time the exploit has been produced, high input h_t has already been generated. The *gain function*, shown in Figure 2, is a function of exploit e and scenario $(h^t, \ell^{t-1}, o^{t-1})$. It yields a real number g representing the success of the exploit.[4] Some prior metrics for QIF consider an exploit to be successful iff the adversary perfectly guesses the secret. But more sophisticated gain functions can quantify the success of the adversary in guessing part of a secret, guessing a secret approximately, or guessing a past secret [14].

D. Example: Password checker

We formalize the password checker from Section II as a system that receives the real password as high input, and a guessed password provided by the adversary as low input. The system produces either `accept` or `reject` as the observable output, depending on whether the guess equals the password. Let $\mathcal{H}$ be the set of real passwords, $\mathcal{L}$ be the set of possible guesses, and $\mathcal{O}$ be $\{\texttt{accept}, \texttt{reject}\}$. At each time step t, it holds that $\Pr(o_t = \texttt{accept}) = 1$ iff $h_t = \ell_t$, and $\Pr(o_t = \texttt{reject}) = 1$ iff $h_t \neq \ell_t$. Each time step models an invocation of the password checker. We do not model the passage of time when there is no guess. If a password change occurs between two guesses, the new password becomes high input to the system when the later guess is provided as low input.

For the high-input strategy, assume that a new password is produced at each time step according to the following password-changing policy:

> A log is maintained of all failed login attempts. From that log, the 10 most frequently guessed passwords are extracted. A log is also maintained of the 5 most recently chosen passwords for each user. When users change their passwords, they must do so in accordance with two rules: (i) a new password cannot coincide with any of the 5 previous passwords; and (ii) a new password cannot coincide with any of the 10 most common guesses.

The high-input strategy, therefore, depends on the history of low inputs and high inputs.

For the action strategy, the adversary produces guesses based on the observation of whether past guesses were successful. Further, there is no need to retry a failed guess until it is likely the password has been changed. So the action strategy depends on the past history of low inputs, and on the past history of observables.

When the adversary has gathered enough information, he can attack by choosing an exploit. The set $\mathcal{E}$ of exploits could simply be the set $\mathcal{L}$ of low inputs, since a natural attack is to try logging in with the password.

E. Probabilities in the presence of feedback

Having added high-input and action strategies to our model, we now need to adapt equation (2) to use them in calculating the probability of executions. Doing so is not straightforward: as shown by Alvim et al. [28], equation (2) does not define a channel that is invariant with respect to the distribution on low and high inputs. Such a channel is required by classic information-theoretic metrics of maximum leakage.

Our solution is to employ a technique proposed by Tatikonda and Mitter [30] and applied by Alvim et al. [28] to interactive systems. The details are given in our technical report [29]. Here, we summarize the two main results.

First, we give a well-defined probability distribution of executions:

Theorem 1. *Given a system and a context, there is a unique joint probability distribution $Q(\eta^T, \alpha^T, h^T, \ell^T, o^T)$ capturing the model of Figure 2.*

The existence of Q allows us to condition the occurrence of system-related events on the occurrence of context-related events, and vice-versa. For example, we can use Q to reason about how much information observables reveal about high-input strategies.

Second, we show that our model's use of deterministic strategies is not a fundamental restriction:

Theorem 2. *Probabilistic strategies can be modeled by probability distributions on deterministic strategies.*

So we can model users and adversaries who use probabilistic strategies. Nonetheless, as we prove below in Theorem 4, there will always be a deterministic action strategy that is optimal against a given high strategy. Clark and Hunt [12] show that, similarly, deterministic strategies suffice to determine whether *input-output labeled transition systems* satisfy noninterference.

IV. Quantifying security

Having defined the full model we can now derive from it means of quantifying security. To facilitate this, we represent our model as a *probabilistic program*, which precisely

[4]The gain function does not have access to future secrets values h_u, where $u > t$. So even if the adversary determines at time t what the high input will be at time $u > t$, the adversary must wait until time u to realize the gain. To give a real-world example, a thief who learns today that a house will be unoccupied next Tuesday still has to wait until next Tuesday to rob the house.

describes the joint distribution it induces. Using this notation makes it easier to define particular scenarios precisely, as is done in Section VI, and maps closely what we do in our implementation. In particular, as Section V describes, we literally implement this model in a probabilistic programming language and use it to compute metrics of interest.

Given this new presentation of the model, we show how to quantify security in terms of the gain adversaries are expected to achieve while interacting with a system. We will describe this expectation in terms of the optimal adversary that strives to achieve the most gain (Section IV-C).[5] In Sections IV-D and IV-E we show this general definition of security expresses *and extends* the existing metrics of vulnerability, g-vulnerability, and guessing entropy.

A. Probabilistic programming

Probabilistic programs permit the expression of probability distributions using familiar programming language notation. In this paper, we will express probabilistic programs in slightly sugared OCaml, an ML-style functional programming language [31]. In essence, probabilistic programs are just normal programs that employ randomness. For example, the following program employs the `random_int` function to draw a random integer from the uniform distribution of non-negative integers (representing the possible real locations of some hidden stash). Each run of the `gen_stash` program can thus be viewed as sampling a number from a uniform distribution of integers between 0 and 7:

```
let gen_stash () =
  let real_loc = (random_int () mod 8) in real_loc
```

Though `gen_stash` is traditionally viewed as sampling values, it can also be seen as a definition of a random variable $X_{\texttt{gen\_stash ()}}$, whose values are uniformly distributed between 0 and 7. We write $X_{\texttt{exp}}$ to denote a random variable whose distribution is defined by the probabilistic program `exp`.

While `gen_stash` takes no arguments, in general functions may take arguments and these arguments might themselves be probabilistic. Consider the following example:

```
let guess (realval: int) (guessval: int) =
  let correct = (realval == guessval) in
    correct
```

The `guess` program takes two arguments and returns whether they are equal. Thus we can define random variable $X_{\texttt{guess (gen\_stash()) 5}}$ over booleans, where **true** will have probability 1/8, and **false** have the remaining probability.

The distribution of this random variable depends on the distribution of the random variable corresponding to `gen_stash ()`, so we can condition the probability of an outcome on the latter given an outcome of the former; e.g.,

$$\Pr\left(X_{\texttt{gen\_stash ()}} \mid X_{\texttt{guess (gen\_stash ()) 5}} = \textbf{false}\right)$$

would be the uniform distribution of integers between 0 and 7, but not including 5.

```
type time = int
type history =
  {t: time; tmax: time;
   highs: ℋ list;
   lows: ℒ list;
   obss: 𝒪 list
   atk: ℰ option}
type 𝒜 = Wait of ℒ | Attack of ℰ

type sysf = history → 𝒪
type highf = history → ℋ
type actf = int → int → ℒ list → 𝒪 list → 𝒜
type gainf = history → float
```

Fig. 3. Types used by the probabilistic program implementing the model.

B. The model as a probabilistic program

Now we consider how to express the model of Figure 2 as a probabilistic program.

Elements of the model: The types of values in the program are $\mathcal{H}$, $\mathcal{L}$, $\mathcal{O}$, and $\mathcal{E}$, as in the information-theoretic presentation. Figure 3 gives the types of other elements used in the model.[6] Define a record type *history* to be the history of execution: the first field of the record contains the current time; the next is the maximum time for the length of the execution of a scenario; the next three contain the high inputs, low inputs, and observations produced thus far; and the last contains the exploit. At each time step, the adversary will produce an *action* $\mathcal{A}$, which is either a low input or an exploit.

Operation of the model: Figure 4 presents the model as a probabilistic program, using two functions. The first, `scenario`, corresponds to the identically named element in Figure 2 while the second, `evaluate`, uses `scenario` and then evaluates the resulting gain from the history produced.

The `scenario` function takes four arguments. The first, `T`, is maximum number of time steps to consider. The second is the system being modeled, which is a function of type *sysf* (all types are defined in Figure 3). The last two arguments comprise the *context*, i.e., the high-input strategy (of type *highf*) and the action strategy (of type *actf*). The scenario starts with the initial history at time 0, with empty lists, and no attack. The loop at line 8 captures the iterations of Figure 2, updating the history for up to `T` iterations, or until the adversary attacks using an exploit. In each iteration, a new secret is produced (Line 11), an adversary action is computed (Line 15), and if the action is to wait, a new observation is made (Line 23).[7] This function returns the full history when it completes.

The `evaluate` function computes how successful the adversary was by applying their exploit to the gain function (Line 36), which has type *gainf*. Evaluation returns minimal gain if the history does not contain an exploit.

Comparing to the information theoretic model: Our probabilistic program corresponds quite closely to the informa-

[5]Our technical report [29] also considers a *memory-limited* adversary.

[6]The type α *option* is an OCaml type whose values are either `None`, or `Some x` where `x` is of type α.

[7]The `@` operator appends two lists. We will use OCaml's array indexing notation `a.(i)` for lists as well. That is, `l.(i) = nth l i`. We also define `last l = l.(length l - 1)` to get the last element of a list.

```
let scenario (T: time) (system: sysf)
  (high_func: highf) (strat_func: actf) =

  let hist = {t = 0; tmax = T; atk = None;
              highs = []; lows = [];
              obss = []} in

  while hist.t <= T && hist.atk = None do
    hist.t <- hist.t + 1;

    let new_high = high_func hist in

    hist.highs <- hist.highs @ [new_high];

    let new_action =
      strat_func hist.t T hist.lows hist.obss in

    match new_action with
    | Attack exp ->
      hist.atk <- Some exp
    | Wait new_low ->
      hist.lows <- hist.lows @ [new_low];
      let new_obs = system hist in
        hist.obss <- hist.obss @ [new_obs]
  done;
  hist

let evaluate (T: time)
          (system: sysf)
          (high_func: highf)
          (gain_func: gainf)
          (strat_func: actf) =
  let hist = scenario T system
                  high_func strat_func in
  let gain = match hist.atk with
    | Some exp -> gain_func hist exp
    | None -> −∞ in
  gain
```

Fig. 4. The model as a probabilistic program.

tion theoretic model of the previous section. The one difference is that the probabilistic program directly employs a single high-input strategy `high_func` that can itself be randomized, as opposed to using a randomized stream of deterministic high-input strategies (and likewise for the action strategy). As per Theorem 2, doing this is still faithful to the model, and it turns out to be more tractable (and convenient) to implement.

C. The general metric

Now we turn our attention to defining a general quantitative metric of information leaked by the model. In what follows we will fix all the `evaluate` parameters except the last two (the gain function and the strategy function). We will use the expression

```
model = evaluate T system high_func
```

as an instantiation of the fixed parameters. The expected gain is thus $\mathbb{E}\left[X_{\texttt{model gain\_func strat\_func}}\right]$.

An information flow metric is most useful when considered from the point of view of a powerful adversary. In fact, we are most interested in what the system can be expected to leak when interacting with an adversary employing the *optimal* strategy.

Definition 3. The *dynamic gain* $\mathbb{D}_{\texttt{gain\_func}}(\texttt{model})$ is the gain an optimal adversary is expected to achieve under the parameters of a `model` from a gain function `gain_func`.

$$\mathbb{D}_{\texttt{gain\_func}}(\texttt{model}) \overset{\text{def}}{=} \max_{\texttt{s}\in \mathit{actf}} \mathbb{E}\left[X_{\texttt{model gain\_func s}}\right]$$

One way to compute the dynamic gain is to consider essentially all adversary choices in the joint distribution describing the model, picking out the best ones. The best choices made thus will both define the optimal adversary, which we call `opt_strat`, and the gain they can expect to achieve. The rest of this subsection considers an algorithm for doing this.

To start, note the adversary's strategy can control three things: the low inputs to the system, when to attack, and the exploit. We introduce all allowed choices via a random strategy that tries everything;[8] its parameters permit modeling adversaries lacking some capabilities.

```
let rand_strat (adaptwait: bool)
               (loworder: ℒ list option): actf =

  fun (t: time) (tmax: time)
      (lows: ℒ list) (obss: 𝒪 list): 𝒜 ->
    if t == tmax || (adaptwait && flip 0.5) then
      Attack (uniform_select [ℰ])
    else match loworder with
      | None -> Wait (uniform_select [ℒ])
      | Some order -> Wait order.(t)
```

This function is parameterized by two things: `adaptwait` that determines whether we are modeling a wait-adaptive adversary and `loworder`, which is `None` for a low-adaptive adversary and otherwise provides a sequence of low inputs. Whenever `adaptwait` is set to **`false`** this strategy only attacks at time `tmax = T` and when it is **`true`** it attempts to attack at any point, randomly, choosing an exploit, randomly again, from $[\mathcal{E}]$, the list of all attacks. If `loworder` is `Some order`, the strategy picks low inputs according to `order` but otherwise picks a low input randomly from $[\mathcal{L}]$, the list of all possible low inputs.

The evaluation of `model gain_func strat` (for some `gain_func` and `strat = rand_strat adaptwait loworder`) produces the random variable $X_{\texttt{model gain\_func strat}}$. Along with the final return value of this expression we will also make use of the random variables for some other expressions that are involved in this computation. Namely, we will make use of the joint distribution over the final `gain` variable, and the `atk`, `highs`, `lows`, and `obss` fields of `hist`:

$$\Pr\left(X_{\texttt{gain}}, X_{\texttt{hist.atk}}, X_{\texttt{hist.highs}}, X_{\texttt{hist.lows}}, X_{\texttt{hist.obss}}\right)$$

To compute the dynamic gain for the optimal adversary, we define two maps *Act* (for action) and *G* (for gain) from lists of low inputs and observations to the optimal adversary's action and expected gain, respectively. Starting from full histories (lists of length `T`) and working backwards to the initial history (empty lists), these maps' construction will define the behavior of strategy `opt_strat` and determine its expected gain. Though the joint distribution is constructed using a random strategy, in

[8]The function `flip p` is a random coin flip returning true or false with probabilities p and $1-p$ respectively. `uniform_select` uniformly picks an element from a given list.

what follows, the probabilities introduced by the strategy will be factored out by conditioning on its output.

When `length lows = length obss = T` we fill in the map G as follows:

$$G[\texttt{lows, obss}] \overset{\text{def}}{=} \max_{\texttt{e}\in\mathcal{E}} G_{\text{attack}}(\texttt{lows,obss,e})$$
$$Act[\texttt{lows, obss}] \overset{\text{def}}{=} \texttt{Attack} \operatorname*{argmax}_{\texttt{e}\in\mathcal{E}} G_{\text{attack}}(\texttt{lows,obss,e}) \tag{3}$$

$$G_{\text{attack}}(\texttt{lows,obss,e}) \overset{\text{def}}{=} \mathbb{E}[X_{\texttt{gain}} \mid X_{\texttt{hist.lows}} = \texttt{lows}, X_{\texttt{hist.obss}} = \texttt{obss}, X_{\texttt{hist.atk}} = \texttt{Some e}]$$

The expression determines the best exploit for an adversary that has chosen the list of lows `lows` and observed `obss` as a result. There is no need to consider a choice of waiting at time `T` as that would result in the minimum gain. As a technicality, we assume that $\mathbb{E}[X_{\texttt{gain}} \mid X] = -\infty$ whenever $\Pr(X) = 0$. This is necessary when modeling non-adaptive adversaries that do not necessarily try attacking at every point (i.e., when the `adaptwait` parameter to `rand_strat` is **`false`**).

For actions before time `T`, the adversary can either attack now, optimizing their exploit `e`, or wait, optimizing their choice of low input `l` for the next observation. Formally, if `length lows = length obss = n < T` we define:

$$G[\texttt{lows, obss}] \overset{\text{def}}{=} \max\{ \max_{\texttt{e}\in\mathcal{E}}\{G_{\text{attack}}(\texttt{lows,obss,e})\}, \overset{\text{def}}{=} B_a \quad \max_{\texttt{l}\in\mathcal{L}}\{G_{\text{wait}}(\texttt{lows,obss,l})\}\} \overset{\text{def}}{=} B_w \tag{4}$$

$$Act[\texttt{lows, obss}] \overset{\text{def}}{=} \begin{cases} \texttt{Attack} \operatorname*{argmax}_{\texttt{e}\in\mathcal{E}} G_{\text{attack}}(\texttt{lows,obss,e}) & \text{if } B_a \geq B_w \\ \texttt{Wait} \operatorname*{argmax}_{\texttt{l}\in\mathcal{L}} G_{\text{wait}}(\texttt{lows,obss,l}) & \text{otherwise} \end{cases}$$

$$G_{\text{wait}}(\texttt{lows,obss,l}) \overset{\text{def}}{=} \mathop{\mathbb{E}}_{\texttt{o}\leftarrow O^{n+}_{\texttt{lows,obss}}(\texttt{l})} G[\texttt{lows @ [l], obss @ [o]}]$$

The outer maximization of Equation 4 describes the adaptive choice of either attacking the system at time `n` or waiting. The optimization of attack is identical to that of Equation 3. Choosing to wait presents the further adaptive choice of the next low input. The notation $O^{n+}_{\texttt{lows,obss}}(\texttt{l})$ in this optimization is a random variable representing the distribution of observables at the next time step given a particular choice `l` of low input.

$$\Pr\left(O^{n+}_{\texttt{lows,obss}}(\texttt{l}) = \texttt{o}\right) \overset{\text{def}}{=} \Pr(X_{\texttt{hist.obss.(n+1)}} = \texttt{o} \mid X_{\texttt{hist.lows}} =_{(n+1)} (\texttt{lows @ [l]}), X_{\texttt{hist.obss}} =_n \texttt{obss}) \tag{5}$$

The $=_n$ notation above corresponds to equality of the first n elements of lists: $X_{\texttt{list1}} =_n \texttt{list2} \overset{\text{def}}{=} X_{\texttt{take n list1}} = \texttt{take n list2}$. The expectation in G_{wait} is due to the fact that the adversary does not know the high part of the history nor has control over the potentially non-deterministic aspects of the system function. When modeling non-low-adaptive adversaries (i.e., when the `loworder` parameter to `rand_strat` is the list of inputs), the strategy does not necessarily try all low inputs, so we need to carefully define $G_{\text{wait}}(\texttt{lows, obss, l}) = -\infty$ whenever $\Pr(X_{\texttt{hist.lows}} = \texttt{lows @ [l]}) = 0$.

Constructing the map G backwards in the length of histories (due to the recursion in the definition of G) eventually produces $G[\texttt{[], []}]$ and this is the expected gain of the optimal attacker in a model, or $\mathbb{D}_{\texttt{gain\_func}}(\texttt{model})$.

Theorem 4. *The maps Act and G define the behavior and expected gain of an optimal adaptive adversary (using `rand_strat` with `waitadapt = true` and `loworder = None`).*

Specifically, let `opt_strat` be defined as follows:

```
let opt_strat: actf =
  fun (t: time) (lows: ℒ list) (obss: 𝒪 list) ->
     Act[lows, obss]
```

Using `opt_strat` achieves the maximal expected gain:

$$\mathbb{D}_{\textit{gain\_func}}(\textit{model}) \overset{\text{def}}{=} \max_{s\in actf} \mathbb{E}[X_{\textit{model gain\_func s}}] = \mathbb{E}[X_{\textit{model gain\_func opt\_strat}}] = G[\textit{[], []}]$$

Proof: (sketch) There are two parts to this claim: (1) that `opt_strat` achieves maximal gain over all strategies, and (2) that this gain is equal to $G[\texttt{[],[]}]$. To show both let us consider the optimal behavior (not necessarily of `opt_strat`) at time `T`, having observed some list of observations `obss` and having chosen low inputs `lows`. The optimal action has to be an exploit as otherwise the gain becomes $-\infty$. The definitions of Equation 3 pick out the exploit `e` which maximizes $G_{\text{attack}}(\texttt{lows,obss,e})$, which by definition is exactly the expected gain achieved by exploiting using `e`. There is no other behavior that does better here, as we have specifically maximized over all options.

We can therefore conclude that $G[\texttt{lows,obss}]$ is indeed the optimal expected gain at time `T` given that `lows` and `obss` have occurred. We then carry this argument backwards to time `T`-1 (and then `T`-2 and so on until time 0). Once again, note the definitions of Equation 4 maximize over all choices the adversary might make. It remains to show that the quantities maximized over are accurate representations of expected gain. The option to attack is based on $G_{\text{attack}}(\texttt{lows,obss})$ which is the same as in the argument for time `T`. For $G_{\text{wait}}(\texttt{lows,obss})$, note that its definition uses the quantities from G for a later time `T` which we presumed are correct. The definition merely performs an expectation over possible observations `o` that might occur at that point. ■

D. Expressing existing metrics

Here we show how our metric for optimal adversary gain subsumes existing metrics, in particular, vulnerability, g-vulnerability, and guessing entropy. We do this in two steps. First, we must use the following *non-wait-adaptive* version of Definition 3 for defining dynamic gain (since classic metrics use non-wait-adaptive adversaries).

Definition 5. The dynamic, not wait-adaptive, gain $\mathbb{D}^{nowait}_{\texttt{gain\_func}}(\texttt{model})$ is the gain an optimal, not wait-adaptive, adversary is expected to achieve under the parameters of a `model` from a gain function `gain_func`.

$$\mathbb{D}^{nowait}_{\texttt{gain\_func}}(\texttt{model}) \stackrel{\text{def}}{=} \max_{\texttt{s} \in actf_{nowait}} \mathbb{E}\left[X_{\texttt{model gain\_func s}}\right]$$

The set $actf_{nowait}$ is the set of all action strategies that only attack at time `T`, hence strategies that are not wait adaptive.

Proposition 1. *Constructing* `opt_strat` *as defined in Section IV-C but using* `rand_strat` *with* `waitadapt = false` *instead yields an action strategy that maximizes the expected gain for a non-wait-adaptive adversary.*

The proof for this proposition essentially follows that of Theorem 4, merely noting that using `rand_strat` with `waitadapt = false` to enumerate possible adversary choices removes exactly those, and no more, that are not available to an adversary who cannot wait.

Now we define a restricted model and a scenario therein that corresponds to the classic scenario, a gain function specific to each metric, and prove that the dynamic not-wait-adaptive gain matches the standard metric. Further details (and proofs) are given in our technical report [29].

The restricted model is defined as follows. First, the high-input strategy is an identity function, since the secret never changes. Second, the gain is defined only in terms of the high value and the exploit (not past observations or low inputs, which are ignored). Additionally, there are no low input choices to make (`type` $\mathcal{L}$ `= unit`) and we use `rand_strat` with `adaptwait = false`, thus eliminating any benefit of low or wait adaptivity. Under these restrictions, Equations 4 and 5 can be rewritten resulting in a simpler definition of gain. In what follows we omit the `lows` parts, and write `secret` to express `last hist.highs`, the sole unchanging high value. We will refer to the expressions `model` and gain functions `gain_func` that instantiate parameters in the restricted manner enumerated here as *static*.

Lemma 1. *If* `model` *and* `gain_func` *are static then the dynamic gain (which would be more appropriately named the static gain) simplifies to the following.*

$$\mathbb{D}^{nowait}_{gain\_func}(model) = \mathop{\mathbb{E}}_{obss \leftarrow X_{hist.obss}} \left[\max_{e \in \mathcal{E}} \mathbb{E}\left(X_{gain} \mid X_{hist.obss} = obss, X_{hist.atk} = Some\ e\right) \right]$$

Now we turn to the particular metrics.

Vulnerability [4]: The notion of vulnerability corresponds to an equality gain function (i.e., a guess).

```
let gain_vul: gainf =
  fun (hist: history) (exp: ℰ) ->
    if secret == exp then 1.0 else 0.0
```

The goal of the attacker assumed in vulnerability is evident from `gain_vul`; they are directly guessing the secret, and they only have one chance to do it.

Theorem 6. *In a static* `model`*, the vulnerability (written* $\mathbb{V}$*) of the secret conditioned on the observations is equivalent to dynamic gain using the* `gain_vul` *gain function.*

$$\mathbb{D}^{nowait}_{gain\_vul}(model) = \mathbb{V}\left(X_{secret} \mid X_{hist.obss}\right)$$

g-vulnerabiity [14]: Generalized gain functions can be used to evaluate metrics in a more fine-grained manner, leading to a metric called g-vulnerability. This metric can also be expressed in terms of the static model. Let `g` be a generalized gain function, returning a *`float`* between 0.0 and 1.0, then we have:

```
let gain_gen_gain (g: ℋ → ℰ → float): gainf =
  fun (hist: history) (exp: ℰ) : float ->
  g secret exp
```

The difference between expected gain and g-vulnerability are non-existent in the static model. The gain of a system corresponds exactly to g-vulnerability of `g`, written $\mathbb{V}_{\texttt{g}}(\cdot)$.

Theorem 7. *In a static* `model` *the g-vulnerability of* `g` *is equivalent to dynamic gain using* `gain_gen_gain g` *gain function.*

$$\mathbb{D}^{nowait}_{gain\_gen\_gain}(model) = \mathbb{V}_{g}\left(X_{secret} \mid X_{hist.obss}\right)$$

Guessing-entropy [23]: Guessing entropy, characterizing the expected number of guesses an optimal adversary will need in order to guess the secret, can also be expressed in terms of the static model. We let attacks be lists of highs (really permutations of all highs). The attack permutation corresponds to an order in which secrets are to be guessed. We then define expected gain to be proportional to how early in that list of guesses the secret appears.

```
type ℰ = ℋ list

let pos_of (secret: ℋ) (exp: ℋ list) =
  (* compute the position of secret in exp *)

let gain_guess_ent: gainf =
  fun (hist: history) (exp: ℰ) =
    -1.0 * (1.0 + (pos_of secret exp))
```

Note that we negate the gain as an adversary would optimize for the minimum number of guesses, not the maximum. Guessing entropy, written $\mathbb{G}$, is related to dynamic gain as follows.

Theorem 8. *In a static* `model`*, guessing entropy is equivalent to (the negation of) dynamic gain using the* `gain_guess_ent` *gain function.*

$$-\mathbb{D}^{nowait}_{gain\_guess\_ent}(model) = \mathbb{G}\left(X_{secret} \mid X_{hist.obss}\right)$$

E. Extending existing metrics

Our model lets us take existing information flow metrics and extend their purview in two directions: temporal variations in the target of the attack and the attacker's capabilities. Each of these extensions can be specified in terms of the more general scenario and gain functions permitted in our model. As such, we preserve the goal of an existing metric but apply it to situations in which the existing definitions are insufficient.

The first direction gives us several choices as to what about the present, past, or future is the intended attack target. We will briefly cover four categories: *moving target*, *specific past*, *historical*, and *change inference*.

1) **Moving target:** The target of the attack is the current high value. Defining the gain in terms of the most recent high value, rather than the original high value, produces the moving-target equivalents of vulnerability, g-vulnerability, and guessing entropy; i.e., we use the same gain functions as Section IV-D but with non-identity high-input strategies.

2) **Specific past gain:** The target of attack is the high value at some fixed point in time (rather than the most recent one). The gain function would thus evaluate success against this secret, independent of the current secret.

3) **Historical gain:** The target of attack is the entire history of high values up to the present time. An attack on the whole history can be formulated by extending the set of attacks to include lists of all lengths up to `T` and specifying dynamic gain in terms of equality of this attack and the history.

4) **Change inference:** The target of the attack is not one or more high values, but rather the high-input strategy that produces them. An adversary's interactions will increase his knowledge of the high-input strategy, but such knowledge is only indirectly quantified by knowledge of the high values themselves. To quantify knowledge of the high-input strategy directly, we could slightly extend the definition of gain functions:

```
type gainf = history → highf → float
```

Due to the difficulty inherent in the problem of determining functional equivalence, such discrimination would need to be syntactic (two semantically identical high-input strategies could be seen as distinct).

On the second axis of extension we have capabilities by which the adversary can interact, or influence, the system prior to attacking. Our model permits consideration of low-adaptive and wait-adaptive adversaries, who can carefully choose how to interact with the system prior to attacking it, and/or wait for the best moment to attack.

Section VI conducts experiments that explore some of these metrics.

V. Implementation

We have implemented our model using a simple monadic embedding of probabilistic computing [32] in OCaml, as per Kiselyov and Shan [33]. The basic approach is as follows. The `model` function, translated into monadic style, is probabilistically evaluated to produce the full joint distribution over the history up to some time `T`. From this joint distribution the optimal adversary's expected gain is constructed according to the algorithm in Section IV-C. The implementation (and experiments from the next section) are available online.[9]

In more detail, our implementation works as follows. We represent a distribution as either a map from values to floats (a slight extension of `PMap` of the extlib library[10]) or an enumerator of value and float tuples (`Enum` of the extlib library), converting between the two at various points when evaluating the program.

```
module M = struct
  type 'a dist = ('a, float) PMap.t
  ...
module E = struct
  type 'a dist = ('a * float) Enum.t
  ...
```

The former is used to represent a mapping of values to their probabilities but requires all these values to be stored in memory. The latter has low memory requirements and is used before the probabilities of values need to be retrieved. These distributions are manipulated in a monadic style using functions such as:

```
val bind: 'a dist -> ('a -> 'b dist) -> 'b dist
val return: 'a -> 'a dist
val bind_flip: float -> (bool -> 'b dist) -> 'b dist
val bind_uniform:
  int -> int -> (int -> 'b dist) -> 'b dist
val bind_uniform_select:
  'a list -> ('a -> 'b dist) -> 'b dist
```

The first two are standard. The next creates a random coin flip and continues the computation over this flip to produce a distribution. The next does the same with a random integer in a given range and the last uniformly picks a value from a list to continue the computation with. The program below shows how these functions are used to compute the probability that the sum of two dice is 10.

```
let d_dice1 = M.bind_uniform 1 6 M.return in
let d_dice2 = M.bind_uniform 1 6 M.return in
let d_sum = M.bind d_dice1 (fun dice1 ->
  M.bind d_dice2 (fun dice2 ->
    M.return (dice1 + dice2))) in
let prob_10 = PMap.find d_sum 10
```

In the first two lines, two distributions are created that map integers 1 through 6 to the probability $1/6$. In the third line `bind` is used to take an initial distribution `d_dice1` and a function that will continue the computation of a distribution starting from a value of the first. This continuation gets called once for each possible value in `d_dice1` and each time produces a distribution of 6 values. All 6 of these distributions are merged into one when this `bind` finishes. The nested `bind` performs a similar task starting with values of `d_dice2`, eventually producing a distribution, `d_sum` over the sum of two dice rolls. The probability of this sum being 10 is looked up at the last line.

To use this approach, functions must be rewritten in monadic style. For example, the rewritten `rand_strat` function (given just below Definition 3, page 6) is the following:

```
let rand_strat (adaptwait: bool)
               (loworder: ℒ list option) =

  fun (t: time) (tmax: time)
      (lows: ℒ list) (obss: 𝒪 list): 𝒜 dist ->
    bind_flip 0.5 (fun flip ->
      if t == tmax || (adaptwait && flip)
        then bind_uniform_select [ℰ]
             (fun exp -> return (Attack exp))
        else match loworder with
          | None -> bind_uniform_select [ℒ]
                    (fun low -> return (Wait low)))
          | Some order -> return order.(t))
```

[9]https://github.com/plum-umd/qif/tree/master/oakland14

[10]ocaml-extlib: https://code.google.com/p/ocaml-extlib

Notice that the output type is now a distribution on actions, not just a single action. Monadic style becomes cumbersome with more complex functions, and though a more direct-style implementation is possible (e.g., as in the second half of Kiselyov and Shan's paper [33]), it did not perform as well.

Our current implementation makes little attempt to optimize the construction of a distribution, and thus is susceptible to a state-space explosion when a scenario has many time steps. Fortunately, probabilistic programming is a growing research area (cf. the survey of Gordon et al. [16]) so we are optimistic that the feasible scale of experiments will increase in the future. Approximate probabilistic inference systems such as those based on graphical models or sampling can be used to estimate gain. Exact implementations based on smarter representations of distributions such as those using algebraic decision diagrams [16] could potentially be used to simulate our model. Additionally, Mardziel et al. [9], [34] show how to soundly approximate a simple metric using probabilistic abstract interpretation; this technique could potentially be extended to also soundly approximate dynamic gain. Presently, our simple implementation proved to be sufficient to demonstrate various aspects of the model on a variety of scenarios.

VI. Experiments

This section describes several experiments we conducted, illustrating the interesting outcomes mentioned in the introduction by measuring the effect of varying different parameters of the model. Our experiments develop several examples on the theme of *stakeouts and raids*, where each example varies different parameters, including whether and how a secret changes, whether and in what manner the adversary is low- and wait-adaptive, and the impact of adding costs to observations. We describe the common elements of the scenario next, and describe each variation we considered in the following subsections.

Suppose an illicit-substance dealer is locked in an ever-persistent game of hiding his stash from the police. The simplest form of this example resembles password guessing, replacing the password with the location of the stash and authentication attempts with "stakeouts" in which police observe a potential stash location for the presence of the stash. After making observations the police will have a chance to "raid" the stash, potentially succeeding. In the meantime the stash location might change.

The stash location will be represented as a integer from 0 to 7, as will be the attacks. Observations will be booleans:

```
type ℋ = ℒ = ℰ = int                    (* 0,...,7 *)
type 𝒪 = bool
```

Stakeouts are carried out by comparing the stakeout location to the real stash location.

```
let stakeout: sysf =
  fun (hist: history) ->
    last hist.highs = last hist.lows
```

For modeling non-low-adaptive adversaries in some of our experiments, we will use a fixed stakeout order by using `rand_strat` with `loworder = Some rotate` where `rotate = [0;...;7;0;...]`.

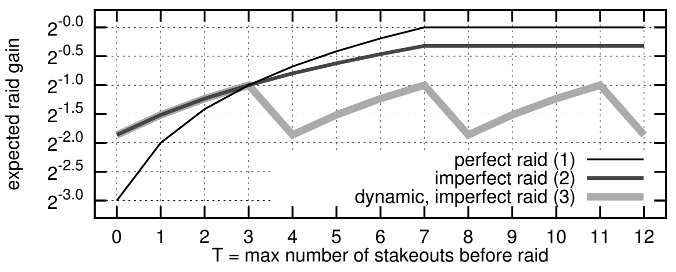


Fig. 5. Expected raid gain over time given stakeouts `stakeout` with (1) static stash `stay_stash` and perfect raids `raid`, (2) static stash and imperfect raids `raid_imperfect`, and (3) moving stash `move_stash 4` and imperfect raids, all with non-adaptive adversaries (`waitadapt = false`, `loworder = Some rotate`).

Raids fail unless the stash and raid locations match:

```
let raid: gainf =
  fun (hist: history) (exp: ℰ) ->
    if last hist.highs = exp then 1.0 else 0.0
```

Finally, for most experiments, the stash moves randomly every 4 time steps (`rate` is 4):

```
let move_stash (rate: int): highf =
  fun (hist: history) ->
    if hist.t mod rate = 0
      then gen_stash ()
      else last hist.highs
```

Above, we reference the `gen_stash` function introduced earlier, which randomly selects a stash location.

```
let gen_stash () =
  let real_loc = (random_int () mod 8) in real_loc
```

The high-input strategy for our final example (Section VI-E) is more complex and will be described later.

A. *How does gain differ for dynamic secrets, rather than static secrets?*

Our first experiment considers the impact on information leakage due to a dynamic secret (using high-input strategy `move_stash`). The adversary here will be non-adaptive and for comparison we also consider a high-input strategy that does not change the secret:

```
let stay_stash: highf =
  fun (hist: history) ->
    if hist.t = 0 then gen_stash ()
               else last hist.highs
```

We also consider a variation of a raid that has a chance to fail even if the raid location is correct, and has a chance to accidentally discover a new stash even if the raid takes place at the wrong location.

```
let raid_imperfect: gainf =
  fun (hist: history) (exp: ℰ) ->
    if last hist.highs = exp
      then flip 0.8
      else flip 0.2
```

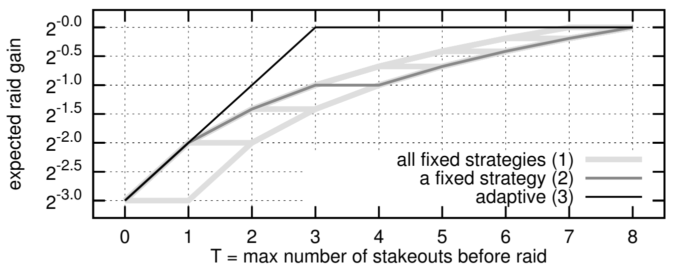


Fig. 6. Expected raid `raid` gain over time with static stash `stay_stash` given stakeouts `stakeout_east_west` with (1) all possible non-adaptive orderings `order` (`loworder = Some order`), (2) a possible non-adaptive ordering (`loworder = Some [0;1;2;7;3;4;5;6]`), and (3) adaptive (`loworder = None`) stakeout locations, all not wait-adaptive (`waitadapt = false`).

Figure 5 plots how the gain differs when we have a static secret with perfect raids, a static secret with imperfect raids, and a dynamic secret with imperfect raids. The static portion (1) with perfect raid is an example of an analysis achievable by a parallel composition of channels and the vulnerability metric [35]. Adding the imperfect gain function (2) alters the shape of vulnerability over time though in a manner that is *not a mere scaling* of the perfect raid case. The small chance of a successful raid at the wrong location results in higher gains (compared to perfect raid) when knowledge is low. With more knowledge, the perfect raid results in more gain than the imperfect. Adding a dynamically changing stash (3) results in a periodic, non-monotonic, gain; though gain increases in the period of unchanging secret, it falls right after the secret changes. This, in effect, is a recovery of uncertainty, which is thus not a non-renewable resource [35]. In the following sections we will refer to the period of time in which the secret does not change as an *epoch*.

B. How does low adaptivity impact gain?

To demonstrate the power of low adaptivity we will use a system function that outputs whether the stash is east or west of the stakeout location. Assuming the stash locations are ordered longitudinally, this function is just a comparison between the stash and stakeout location. The non-adaptive adversary will pick the stakeouts in a fixed order specified by letting `loworder = Some order` for some ordering of low inputs `order`, whereas the low-adaptive adversary will use `loworder = None`:

```
let stakeout_east_west: sysf
  fun (hist: history) ->
    last hist.highs <= last hist.lows
```

Figure 6 demonstrates the expected gain of both types of attackers. For fixed-order (non-adaptive) adversaries, we used all 8! permutations of the 8 stash locations as possible orders and plotted them all as the wide light gray lines in the figure. Though there are many possible orders, the only thing that makes any difference in the gain over time is the position of 7 (the highest stash location) in the ordering as the system function for this input reveals no information whatsoever. All other stakeout locations reveal an equal amount of information in terms of the expected gain. To demonstrate this behavior, we have specifically plotted in the figure the gain for an ordering in which location 7 is staked-out at time 3 (labeled “a fixed strategy”). Gain increases linearly with every non-useless observation. On the other hand, the low-adaptive adversary performs binary search, increasing his gain exponentially.

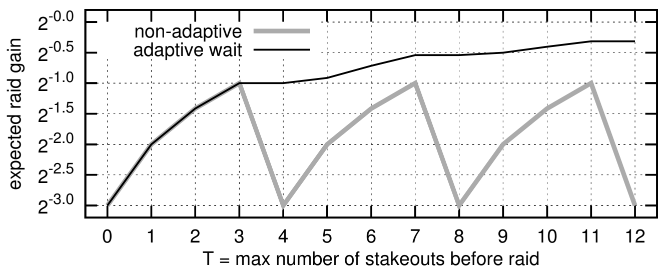


Fig. 7. Expected `raid` gain with moving stash `move_stash 4` given stakeouts `stakeout` with (1) non-wait-adaptive adversary (`waitadapt = false`) and (2) wait-adaptive adversary (`waitadapt = true`), all not low-adaptive (`loworder = Some rotate`).

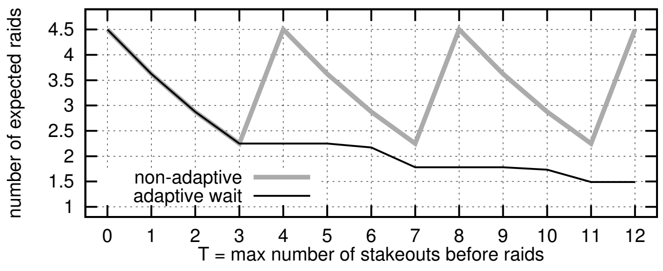


Fig. 8. Expected number of raids (gain of `raid_guess`) with moving stash `move_stash 4` given stakeouts `stakeout` with (1) non-wait-adaptive adversary (`waitadapt = false`) and (2) wait-adaptive adversary (`waitadapt = true`), all not low-adaptive (`loworder = Some rotate`).

C. How does wait adaptivity impact gain?

An adversary that can wait is allowed to attack at any time. Adaptive wait has a significant impact on the gain an adversary might expect. In the simple stakeout/raid example of Figure 7, it transforms an ever-bounded vulnerability to one that steadily increases in time. Using the `raid_guess` function given below we can compute the guessing entropy, which the experiment in Figure 8 shows will steadily decrease (recall from Section IV-D that guessing entropy is inversely related to dynamic gain).

```
type ℰ = ℋ list

let raid_guess: gainf =
  fun (hist: history) (exp: ℰ) ->
    -1.0*(1.0+(pos_of (last hist.highs) exp))
```

Roughly, the optimal behavior for a wait-adaptive adversary is to wait until a successful stakeout before attacking. The more observations there are, the higher the chance this will occur. This results in the monotonic trend in gain over time.

There are subtle decisions the adversary makes in order to determine whether to wait and allow the secret to change. For example, in Figure 7, if the adversary has to attack at time 5 or earlier and has not yet observed a successful stakeout by time 3, they will wait until time 5 to attack, letting all their accumulated knowledge be invalidated by the change that occurs at time 4. This seems counter-intuitive as their odds of a successful raid at time 3 is $1/5$ (they eliminated 3 stash locations from consideration), whereas they will accumulate only 1 relevant stakeout observation by time 5. Having 3 observations seems preferable to 1. The optimal adversary will wait because the *expected* gain at time 5 is actually better than $1/5$; there is $1/8$ chance that the stakeout at time 5 will pinpoint the stash, resulting in gain of 1, and $7/8$ chance it will not, resulting in expected gain of $1/7$. The expectation of gain is thus $^1/_8 \cdot 1 + {}^7/_8 \cdot {}^1/_7 = {}^1/_4 > {}^1/_5$. The adversary thus has better expected gain if they have to attack by time 5 as compared to having to attack by time 3 or 4, despite having to raid with only one observation's worth of knowledge. One can modify the parameters of this experiment so that this does not occur, forcing the adversary to attack before the secret changes. This results in a longer period of constant vulnerability after each stash movement.

This ability to wait is the antithesis of moving-target vulnerability. Though a secret that is changing with time serves to keep the vulnerability at any fixed point low, the vulnerability *for some point* can only increase with time. The drug dealer of our running example would be foolhardy to believe that he is safe from a police raid just because it is unlikely to happen on Wednesday, or any other fixed day; if stakeouts continue and the police are smart enough to not schedule drug busts before having performed the surveillance, the dealer will be caught. On the other hand, if there is a high enough cost associated with making an observation, the vulnerability against raids can be very effectively bounded (as we show in the next experiment).

In fact, the monotonically increasing vulnerability is a property of any scenario where the adversary can decide when to attack.

Theorem 9. *Given any gain function* `g` *that is invariant in* T *(the value* `hist.tmax`*), we have* $\mathbb{D}_{g}($`evaluate (t+1) ...`$) \geq \mathbb{D}_{g}($`evaluate t ...`$)$.

Proof: (Sketch) The theorem holds as an adversary attacking a system with $T = t{+}1$ will have a chance to attack at time t, using the same exact gain function that the adversary would attack were $T = t$, and using the same inputs. We assumed the gain functions are invariant in T hence their gains must be identical. Naturally in the first situation, the attacker can also wait if the expectation of gain due to waiting one more time step is higher. ■

D. Can gain be bounded by costly observations?

Our model is general enough to express costs associated with observations. For example, we can modify our scenario so that the police either observe nothing (indicated by low input and output `None`), or perform a stakeout.

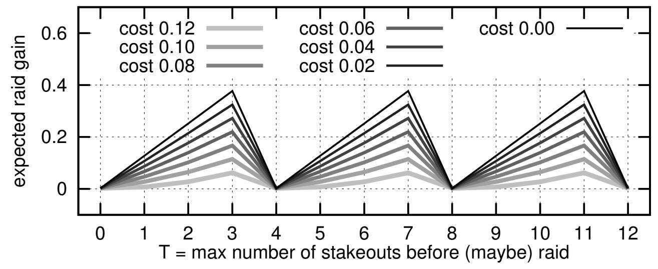
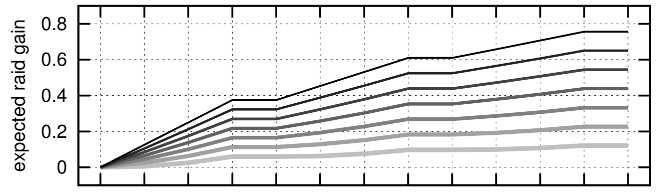



Fig. 9. Expected `raid_option` gain with costly stakeouts `stakeout_option` and moving stash `move_stash 4` with (top) non-wait-adaptive adversaries (`waitadapt = false`) and (bottom) wait-adaptive adversaries (`waitadapt = true`), all not low-adaptive (`loworder = Some rotate`).

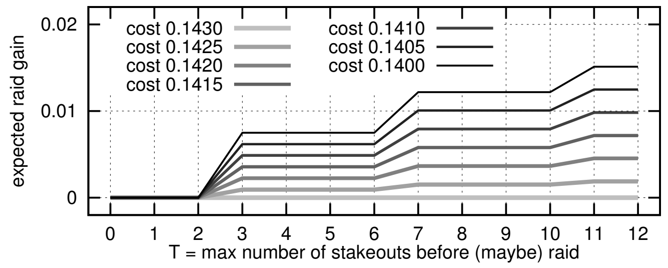


Fig. 10. (Zoomed in) expected `raid_option` gain with costly stakeouts `stakeout_option` and moving stash `move_stash 4` with (top) non-wait-adaptive adversaries (`waitadapt = false`) and (bottom) wait-adaptive adversaries (`waitadapt = true`), all not low-adaptive (`loworder = Some rotate`).

```
type ℒ = ℰ = int option         (* int in 0,...,7 *)
type 𝒪 = bool option
```

```
let stakeout_option: sysf =
  fun (hist: history) ->
    match last hist.lows with
     | None -> None
     | Some stakeout ->
       Some (last hist.highs = stakeout)
```

However, each stakeout performed will have a cost that is applied to the final gain (`c` per stakeout). Additionally, the police are penalized -1.0 units for a raid on the wrong place, and have the option of not raiding at all (for 0.0 gain).

```
let raid_option (cost: float): gainf =
  fun (hist: history) (exp: ℰ) ->
    let raid_gain =
      match exp with
        | None -> 0.0
        | Some raid_loc ->
          if last hist.highs = raid_loc
            then 1.0
            else -1.0 in
    let stakeouts = (count_some hist.lows) in
        (* count how many low inputs in
           hist.lows were not None *)
      raid_gain - cost * stakeouts
```

The results of this scenario are summarized in Figure 9 for stakeout costs ranging from 0.00 to 0.12 and in Figure 10 for higher costs in the range 0.1400 to 0.1430. In the top half of the first figure, the police do not have a choice of when to attack and the optimal behavior is to only perform stakeouts in the epoch that the raid will happen, resulting in the periodic behavior seen in the figure. Higher costs scale down the expected gain.

For wait-adaptive adversaries the result is more interesting. For sufficiently small stakeout costs, the adversary will keep performing stakeouts at all times except for the time period right before the change in the stash location and attack only when the stash is pinpointed. This results in the temporary plateau every 4 steps seen in the bottom half of Figure 9. This behavior seems counter-intuitive as one would think the stakeout costs will eventually make observations prohibitively expensive. Every epoch, however, is identical in this scenario, the stash location is uniform in $0, ..., 7$ at the start, and the adversary has 4 observations before it gets reset. If the optimal behavior of the adversary in the first epoch is to stakeout (at most 3 times), then it is also optimal for them to do it during the second (if they have not yet pinpointed the stash). It is still optimal on the $(n+1)^{th}$ epoch after failures in the first n. As it is, the expectation of gain due to 3 stakeouts is higher than no guesses in any epoch, despite the costs.

The optimal behavior is slightly different for high enough stakeout costs but the overall pattern remains the same. Figure 10 shows the result of an adversary staking out in an epoch only if he is allowed enough time to attack at the end of the epoch. That is, for `T` equal to 1 or 2, the police would not stakeout at all, but if `T` is 3, they will stakeout at times 1,2, and 3. This is because the expected gain given 1 or 2 stakeouts is lower than 0, but for 3 stakeouts, it is greater than 0. As was the case with the lower cost, since the expected gain of observing in an epoch (3 times) is greater than not, the optimal adversary will continue to observe indefinitely if he is given the time.

Note however, that observing indefinitely in these examples does not mean that the expected gain approaches 1. Analytical analysis of this scenario tells us that after n full epochs, the expected gain is $1/8(3-21c)\sum_{i=0}^{n-1}(5/8)^i$ and in the limit, this quantity approaches $1-7c$. The adversary's gain is thus bounded by $1-7c$ for varying stakeout costs c (the point at which it is optimal to not observe at all is $c = 1/7 \approx 0.1429$).

The fact that the adversary's gain can be bounded arbitrarily close to 0, despite their strategy of performing stakeouts indefinitely, highlights a shortcoming of our present model: the assumption of zero-sum relationship between adversary and system. By quantifying optimal adversary's gain, we are implicitly assuming that their gain is our loss. This, however, is hard to justify in some situations like the example of this section. Under the optimal adversary, the drug dealer's stash will be discovered despite the bounded expected police gain from said discovery. Ideally the drug dealer's gain should be a different function than the negation of the police's gain. This idea, and the more game theoretic scenario it suggests, forms part of our ongoing work.

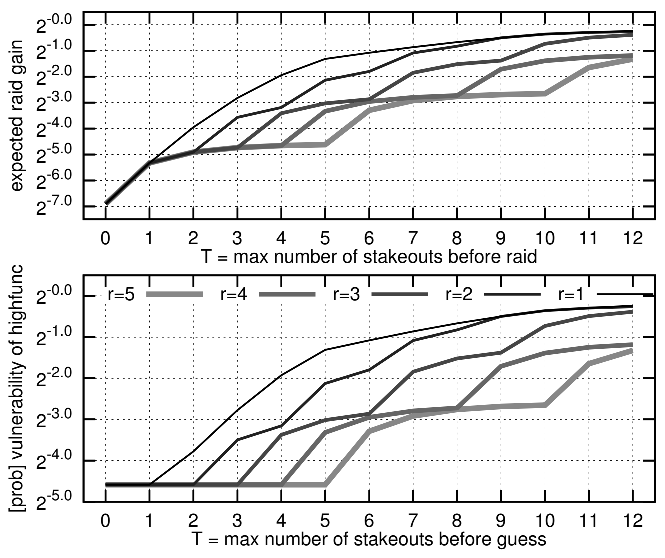


Fig. 11. Changing stash according to `gangs_move` with stakeouts `stakeout_building` quantifying (top) expected raid `raid_apartment` gain and (bottom) expected `high_func` vulnerability, all with wait-adaptive adversaries (`waitadapt = true`) and no low choices (**type** $\mathcal{L}$ = *unit*).

E. Does more frequent change necessarily imply less gain?

In our last example, we demonstrate a counter-intuitive fact that more change is not always better. So far we have used a high-input strategy that is known to the attacker but here he will only know the distribution over a set of possible strategies. Furthermore, guessing the full secret will *require* knowledge of the high-input strategy and therefore will require change to occur sufficiently many times.

Let `nbuilds` be the number of buildings (in the example figure, `nbuilds` will be 5), and `nfloors = factorial (nbuilds-1)` be the number of floors in each of these buildings. The (`nbuilds`) shady buildings are in a city where illicit stashes tend to be found. Each building has `nfloors` floors and each floor of each building is claimed by a drug-dealing gang. A single gang has the same-numbered floor in all the buildings. That is, gang 0 will have floor 0 claimed in every building, gang 1 will have floor 1 in every building, and so on.

The police know there is a stash hidden on some floor in some building and that every gang moves their stash once in a while from one building to another in a predictable pattern (the floor does not change). They know all `nbuilds` gangs each have a unique permutation π of $0, ...,$ `nbuilds-1` as their stash movement pattern. The police also know which floors belong to which gangs (and their permutation). Given `r` as the stash movement rate parameter, the movement of the stash is governed by the following function:

```
type H = {building: int; floor: int}
   (* building int in 0,...,4 *)
   (* floor int in 0,...,23 *)

let gang_move: highf =
  let gang = (random_int ()) mod nfloors in
  let π = (gen_all_permutations nbuilds).(gang) in

  fun (hist: history) ->
```

```
    let c_floor = (last hist.highs).floor in
    let c_build = (last hist.highs).building in
    if hist.t > 0
    then begin
      if t mod r = 0
        then {building = π (c_build);
              floor    = c_floor}
        else stash
    end else
      { building = (random_int ()) mod nbuilds;
        floor    = c_floor }

    in gang_move
```

The function first picks a random gang and generates the permutation that represents that gang's stash movement pattern. It then creates a high-input strategy that will perform that movement, keeping the floor the same, while moving from building to building (the function picks a random building at time 0).

The police set up stakeouts to observe all the buildings but are only successful at detecting activity half the time, and they cannot tell on which floor the stash activity takes place, just which building:

```
type ℒ = unit
type 𝒪 = int option           (* int in 0,...,4 *)

let stakeout_building: sysf =
  fun (hist: history) ->
    if flip 0.5
      then Some (last hist.highs).building
      else None
```

The police want to raid the stash but cannot get a warrant for the whole building, they need to know the floor too:

```
type ℰ = ℋ

let raid_apartment: gainf =
  fun (hist: history) (exp: ℰ) ->
    let build = (last hist.highs).building in
    let floor = (last hist.highs).floor in
    if build = exp.building &&
       floor = exp.floor
    then 1.0 else 0.0
```

Now, the chances of a successful police raid after a varying number of stakeouts depends on the stash change rate `r`. Unintuitively, frequent stash changes lead the police to the stash more quickly. Figure 11(top) shows the gain in the raid after various number of stakeouts, for four different stash change rates (`nbuilds` = 5). The chances of a failed observation in this example are not important and are used to demonstrate the trend in gain over a longer period of time. Without this randomness, the gains quickly reach 1.0 after $\texttt{r} * (\texttt{nbuilds} - 1)$ observations (exactly enough to learn the initially unknown permutation).

The example has a property that the change function (the permutation π of the gang) needs to be learned in order to determine the floor of the stash accurately. Observing infinitely many stakeouts of the same building would not improve police's chance beyond 1 in `nfloors`; learning the high-input strategy here absolutely requires learning how it changes the secret. One can see the expected progress in learning the high-input strategy in Figure 11(bottom), note the clear association between knowing the strategy and knowing the secret. We theorize that this correlation is a necessary part of examples that have the undesirable property that more change leads to more vulnerability. Characterizing this for scenarios in general is another part of our ongoing work.

VII. Related Work

Other works in the literature have considered systems with some notion of time-passing. Massey [36] considers systems that can be re-executed several times, whereby new secret and observable values are produced constantly. He conjectured that the flow of information in these systems is more precisely quantified by *directed information*, a form of Shannon entropy that takes causality into consideration, which was later proved correct by Tatikonda and Mitter [30]. Alvim et al. use these works to build a model for *interactive systems* [28], in which secrets and observables may interleave and influence each other during an execution. The main differences between their model and ours are: (i) they see the secret growing with time, rather than evolving; (ii) they consider Shannon-entropy as a metric of information, rather than vulnerability metrics; and (iii) they only consider passive adversaries.

Köpf and Basin [15] propose a model for adaptive attacks on deterministic systems, and show how to calculate bounds on the information leakage of such systems. Our model generalizes theirs in that we consider probabilistic systems. Moreover, we distinguish between the adversarial production of low inputs and of exploits, and allow adversaries to wait until the best time to attack, based on observations of the system, which itself could be influenced by the choice of low inputs.

In the context of Location Based Services, the privacy of a moving individual is closely related to the amount of time he spends in a certain area, and Marconi et al. [37] demonstrate how an adversary with structured knowledge about a user's behavior across time can pose a direct threat to his privacy.

The work of Shokri et al. [38] strives to quantify the privacy of users of location-based services using Markov models and various machine learning techniques for constructing and applying them. Location privacy is a useful application of our framework, as a principal's location may be private, and evolves over time in potentially predictable ways. Shokri et al.'s work employs two phases, one for learning a model of how a principal's location could change over time, and one for de-anonymizing subsequently observed, but obfuscated, location information using this model. Our work focuses on information theoretic characterizations of security in such applications, and permits learning the change function and the secrets in a continual, interleaved (and optimized) fashion. That said, Shokri et al's simpler model and approximate techniques allows them to consider more realistic examples than those described in our work. Subsequent work [39] considers even simpler models (e.g., that a user's locations are independent).

Classic models of QIF in general assume that the secret is fixed across multiple observations, whereas we consider dynamic secrets that evolve over time, and that may vary as the system interacts with its environment. Some approaches capture interactivity by encoding it as a single "batch job" execution of the system. Desharnais et al. [27], for instance, model the system as a channel matrix of conditional probabilities of whole output traces given whole input traces. Besides creating technical difficulties for computing maximum leakage [28], this approach does not permit low-adaptive or wait-adaptive adversaries, because it lacks the feedback loop present in our model.

O'Neill et al. [13], based on Wittbold and Johnson [11], improve on batch-job models by introducing strategies. The strategy functions of O'Neill et al. are deterministic, whereas ours are probabilistic. And their model does not support wait-adaptive adversaries. So our model of interactivity subsumes theirs.

Clark and Hunt [12], following O'Neill et al., investigate a hierarchy of strategies. *Stream strategies*, at the bottom of the hierarchy, are equivalent to having agents provide all their inputs before system execution as a stream of values. So with stream strategies, adversaries must be non-adaptive. Clark and Hunt show that, for deterministic systems, noninterference against a low-adaptive adversary is the same as noninterference against a non-adaptive adversary. This result does not carry over to quantification of information flow; low-adaptive adversaries derive much more gain than non-adaptive ones as seen in Section VI-B. At the top of Clark and Hunt's hierarchy, strategies may be nondeterministic, whereas our model's are probabilistic. Probabilistic choice refines nondeterministic choice [40], so in that sense our model is a refinement of Clark and Hunt's. But probabilities are essential for information-theoretic quantification of information flow. Clark and Hunt do not address quantification, instead focusing on the more limited problem of noninterference. Nor does their model support wait-adaptive adversaries. There are other, nonessential differences between our model and Clark and Hunt's models: Clark and Hunt allow a lattice of security levels, whereas we allow just high and low; our model only produces low outputs, whereas theirs permits outputs at any security level; and our computation model is probabilistic automata, whereas theirs is labeled transition systems.

VIII. Conclusions and Future Work

In this paper we have presented a new model for quantifying the information flow of a system whose secrets evolve over time. Our model involves an adaptive adversary, and characterizes the costs and benefits of attacks on the system. We showed that an adaptive view of an adversary is crucial in calculating a system's true vulnerability which could be greatly underestimated otherwise. We also showed that though adversary uncertainty can effectively be recovered if the secret changes, if the adversary can adaptively wait to attack, vulnerability can only increase in time. Also, contrary to intuition, we showed that more frequent changes to secrets can actually make them more vulnerable.

Our future work has three main thrusts. Firstly we are generalizing our model further to give the user non-fixed decisions and goals distinct from those of the adversary. Doing so will let us better model examples like that of Section VI-D where adversary gain does not quite mirror the loss of the user. Furthermore, to handle scenarios in which the user and adversary do not fully observe each other's decisions, or when they make decisions simultaneously, we are looking into a game-theoretic analysis of the problem using Bayesian games and their equilibria.

The second avenue of our future work is information theoretical characterizations of various phenomena present in our model which we have hinted at in this paper:

- In Section VI-B we saw that the impact of low-adaptive adversaries range from none to an exponential difference in gain. It is easier, however, to analyze non-adaptive adversaries. It would be valuable to be able to tell when ignoring low-adaptivity will not have a significant impact on the resulting analysis. This would in effect be the quantified version of the theorem of Clark and Hunt [12] which states that non-adaptive strategies are sufficient to ascertain non-interference in deterministic systems.
- In Section VI-E we saw how changing the secret more often is not always preferable to changing it less. We conjectured that such situations require a strong correlation between the secret and the high-input strategy used to evolve the secret. Precisely characterizing this correlation and the contexts in which it is relevant would be useful for building more robust systems.
- The analyses in the paper are framed in the context of a system. Unfortunately, this context directly influences how much information is leaked, so that the less that is known about the context, the less we can say about the security of the system. Prior works attempt to speak of the worst-case leakage for all possible contexts, but for us contexts are richer in structure, making such analysis more difficult. We consider such worst-case reasoning challenging future work.

Finally we are bringing in the works on approximate but sound probabilistic programming [9], [34] to enable the simulation of larger and more complex scenarios. Though these works are only concerned with a metric similar to vulnerability, it may be possible to extend the approach to soundly approximate dynamic gain as well. Additionally exact probabilistic programming systems with smarter representations of distributions such as algebraic decision diagrams in [16] could potentially be applied to our model. We are also investigating this possibility.

Acknowledgements: We thank Mudhakar Srivatsa and Andre Scedrov for useful discussions about this work. We also gratefully thank the anonymous reviewers and our shepherd for their helpful feedback and comments.

For Mardziel and Hicks, this research was sponsored by US Army Research laboratory and the UK Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Army Research Laboratory, the U.S. Government, the UK Ministry of Defense, or the UK Government. The US and UK Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon. Alvim was supported in part from the AFOSR MURI "Science of Cyber Security: Modeling, Composition, and Measurement" as AFOSR grant FA9550-11-1-0137. Much of this research was conducted while Alvim was a postdoctoral research associate in the Mathematics Department at the University of Pennsylvania under the supervision of Prof. Andre Scedrov, whom we gratefully acknowledge. Clarkson was supported in part by AFOSR grant FA9550-12-1-0334.

References

[1] I. S. Moskowitz, R. E. Newman, and P. F. Syverson, "Quasi-anonymous channels," in *Proc. of CNIS*. IASTED, 2003, pp. 126–131.

[2] D. Clark, S. Hunt, and P. Malacaria, "Quantitative information flow, relations and polymorphic types," *J. of Logic and Computation*, vol. 18, no. 2, pp. 181–199, 2005.

[3] K. Chatzikokolakis, C. Palamidessi, and P. Panangaden, "Anonymity protocols as noisy channels," *Inf. and Comp.*, vol. 206, no. 2–4, pp. 378–401, 2008. [Online]. Available: http://hal.inria.fr/inria-00349225/en/

[4] G. Smith, "On the foundations of quantitative information flow," in *Proceedings of the Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, 2009.

[5] M. R. Clarkson, A. C. Myers, and F. B. Schneider, "Quantifying information flow with beliefs," *Journal of Computer Security*, vol. 17, no. 5, pp. 655–701, 2009.

[6] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic discovery and quantification of information leaks," in *IEEE Security and Privacy*, 2009.

[7] C. Mu and D. Clark, "An interval-based abstraction for quantifying information flow," *Electron. Notes Theor. Comput. Sci.*, vol. 253, pp. 119–141, November 2009.

[8] B. Köpf and A. Rybalchenko, "Approximation and randomization for quantitative information-flow analysis," in *CSF*, 2010.

[9] P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa, "Dynamic enforcement of knowledge-based security policies," in *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*, 2011.

[10] R. Segala, "Modeling and verification of randomized distributed real-time systems," Ph.D. dissertation, Massachusetts Institute of Technology, Jun. 1995, tech. Rep. MIT/LCS/TR-676.

[11] J. T. Wittbold and D. Johnson, "Information flow in nondeterministic systems," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 1990, pp. 144–161.

[12] D. Clark and S. Hunt, "Non-interference for deterministic interactive programs," in *Proceedings of the Workshop on Formal Aspects in Security and Trust*, 2008, pp. 50–66.

[13] K. R. O'Neill, M. R. Clarkson, and S. Chong, "Information-flow security for interactive programs," in *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*, 2006, pp. 190–201.

[14] M. S. Alvim, K. Chatzikokolakis, C. Palamidessi, and G. Smith, "Measuring information leakage using generalized gain functions," in *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*, 2012.

[15] B. Köpf and D. Basin, "An information-theoretic model for adaptive side-channel attacks," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2007.

[16] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani, "Probabilistic programming," in *International Conference on Software Engineering (ICSE, FOSE track)*, 2014. [Online]. Available: http://research.microsoft.com/pubs/208585/fose-icse2014.pdf

[17] D. Denning, *Cryptography and Data Security*. Reading, Massachusetts: Addison-Wesley, 1982.

[18] J. Y. Halpern, *Reasoning about Uncertainty*. Cambridge, Massachusetts: MIT Press, 2003.

[19] P. Malacaria, "Assessing security threats of looping constructs," in *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, M. Hofmann and M. Felleisen, Eds. ACM, 2007, pp. 225–235. [Online]. Available: http://doi.acm.org/10.1145/1190216.1190251

[20] P. Malacaria and H. Chen, "Lagrange multipliers and maximum information leakage in different observational models," in *Proceedings of the 2008 Workshop on Programming Languages and Analysis for Security (PLAS 2008)*, Úlfar Erlingsson and Marco Pistoia, Ed. Tucson, AZ, USA: ACM, June 2008, pp. 135–146.

[21] I. S. Moskowitz, R. E. Newman, D. P. Crepeau, and A. R. Miller, "Covert channels and anonymizing networks." in *Workshop on Privacy in the Electronic Society 2003*, 2003, pp. 79–88.

[22] M. S. Alvim, M. E. Andrés, and C. Palamidessi, "Information Flow in Interactive Systems," in *Proceedings of the 21th International Conference on Concurrency Theory (CONCUR 2010), Paris, France, August 31-September 3*, ser. Lecture Notes in Computer Science, P. Gastin and F. Laroussinie, Eds., vol. 6269. Springer, 2010, pp. 102–116. [Online]. Available: http://hal.archives-ouvertes.fr/inria-00479672/en/

[23] Massey, "Guessing and entropy," in *Proceedings of the IEEE International Symposium on Information Theory*. IEEE, 1994, p. 204.

[24] P. Malacaria, "Algebraic foundations for information theoretical, probabilistic and guessability measures of information flow," *CoRR*, vol. abs/1101.3453, 2011.

[25] Pliam, "On the incomparability of entropy and marginal guesswork in brute-force attacks," in *Proceedings of INDOCRYPT: International Conference in Cryptology in India*, ser. Lecture Notes in Computer Science, no. 1977. Springer-Verlag, 2000, pp. 67–79.

[26] C. Braun, K. Chatzikokolakis, and C. Palamidessi, "Quantitative notions of leakage for one-try attacks," in *Proceedings of the 25th Conf. on Mathematical Foundations of Programming Semantics*, ser. Electronic Notes in Theoretical Computer Science, vol. 249. Elsevier B.V., 2009, pp. 75–91. [Online]. Available: http://hal.archives-ouvertes.fr/inria-00424852/en/

[27] J. Desharnais, R. Jagadeesan, V. Gupta, and P. Panangaden, "The metric analogue of weak bisimulation for probabilistic processes," in *LICS*, 2002, pp. 413–422.

[28] M. S. Alvim, M. E. Andrés, and C. Palamidessi, "Quantitative information flow in interactive systems," *Journal of Computer Security*, vol. 20, no. 1, pp. 3–50, 2012.

[29] P. Mardziel, M. S. Alvim, M. Hicks, and M. R. Clarkson, "Quantifying information flow for dynamic secrets," University of Maryland Department of Computer Science, Tech. Rep. CS-TR-5035, 2014, (extended technical report).

[30] S. Tatikonda and S. K. Mitter, "The capacity of channels with feedback," *IEEE Transactions on Information Theory*, vol. 55, no. 1, pp. 323–349, 2009.

[31] "The Caml language," http://caml.inria.fr.

[32] N. Ramsey and A. Pfeffer, "Stochastic lambda calculus and monads of probability distributions," in *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages (POPL)*, 2002.

[33] O. Kiselyov and C. chieh Shan, "Embedded probabilistic programming," in *Proceedings of the Working Conference on Domain Specific Languages (DSL)*, 2009.

[34] P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa, "Dynamic enforcement of knowledge-based security policies using abstract interpretation," *Journal of Computer Security*, vol. 21, no. 4, pp. 463–532, 2013.

[35] B. Espinoza and G. Smith, "Min-entropy as a resource," in *Information and Computation*, 2013.

[36] J. L. Massey, "Causality, feedback and directed information," in *Proc. of the 1990 Intl. Symposium on Information Theory and its Applications*, November 1990.

[37] L. Marconi, R. D. Pietro, B. Crispo, and M. Conti, "Time warp: How time affects privacy in lbss," in *ICICS*, 2010, pp. 325–339.

[38] R. Shokri, G. Theodorakopoulos, J.-Y. L. Boudec, and J.-P. Hubaux, "Quantifying location privacy," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2011.

[39] R. Shokri, G. Theodorakopoulos, C. Troncoso, J.-P. Hubaux, and J.-Y. L. Boudec, "Protecting location privacy: optimal strategy against localization attacks," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012, pp. 617–627.

[40] A. McIver and C. Morgan, *Abstraction, Refinement and Proof for Probabilistic Systems*. New York: Springer, 2005.