Predictive Mitigation of Timing Channels in Interactive Systems

Danfeng Zhang
zhangdf@cs.cornell.edu

Aslan Askarov
aslan@cs.cornell.edu

Andrew C. Myers
andru@cs.cornell.edu

Department of Computer Science
Cornell University
Ithaca, NY 14853

Abstract

Timing channels remain a difficult and important problem for information security. Recent work introduced predictive mitigation, a new way to mitigating leakage through timing channels; this mechanism works by predicting timing from past behavior, and then enforcing the predictions. This paper generalizes predictive mitigation to a larger and important class of systems: systems that receive input requests from multiple clients and deliver responses. The new insight is that timing predictions may be a function of any public information, rather than being a function simply of output events. Based on this insight, a more general mechanism and theory of predictive mitigation becomes possible. The result is that bounds on timing leakage can be tightened, achieving asymptotically logarithmic leakage under reasonable assumptions. By applying it to web applications, the generalized predictive mitigation mechanism is shown to be effective in practice.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*

General Terms

Security

Keywords

Timing channels, mitigation, interactive systems, information flow

1. Introduction

The time at which a computing system performs some observable action such as sending a network packet can in principle encode an unbounded amount of information about what is happening inside the system, creating a *timing channel* [1]. An adversary able to accurately measure this time may learn confidential information from this side channel (e.g., [2, 3, 4, 5]); an adversary able to influence this time may additionally use it as a covert channel to communicate confidential information (e.g., [6, 7, 8]).

Though the recent work cited above demonstrates the threat of timing channels, controlling them without compromising functionality is typically considered to be extremely challenging or even infeasible [9, 10, 11]. Recent work on timing channels has focused on quantitatively bounding what can be learned from timing channels rather than on blocking them entirely (e.g., [12, 13, 14])

Recent work by Askarov et al. introduced a new mechanism called *predictive mitigation* for bounding information leakage via timing channels [14]. Unlike work focusing on preventing leakage of keys from cryptographic operations (such as [12, 13]), predictive mitigation applies to any computing system, making few assumptions about the nature of the computation being performed. However, as we argue, the original predictive mitigation mechanism is impractical for many real-world systems where timing channels are of concern—especially networked servers such as web applications. Therefore, this paper generalizes predictive mitigation to take advantage of more knowledge about the system whose timing channels are being mitigated, significantly improving the tradeoff between security and performance.

Contributions. The contributions of this work are both theoretical and practical. On the theoretical side, the theory of predictive mitigation is extended in several ways:

• **Inputs.** The model of the mitigated system is extended to account for inputs to the system, so output timing can be predicted from public (that is, nonconfidential) attributes of input such as request time.

• **Threads.** In [14], the system being mitigated is a black box. Here the system is modeled more concretely as containing multiple threads which communicate with the outside over different output channels. This more detailed modeling enables tighter leakage bounds.

• **Composition.** In general, a system employing predictive mitigation may be composed of several communicating components, each individually mitigated. The theory of composing predictive mitigation is developed.

This new theory of predictive mitigation has been put into practice as in an implementation of predictive mitigation for web applications. For example, we implement a standardized server-side wrapper that can mitigate timing leaks from any web application.

An important contribution of this paper is an empirical evaluation of how predictive mitigation performs when applied to real applications with different characteristics. We examine its impact on latency, throughput, and maximum timing leakage of wrapped web applications. The results from this implementation suggest that the generalized predictive mitigation mechanism appears to be practical and offers a significant improvement on the original predictive mitigation method.

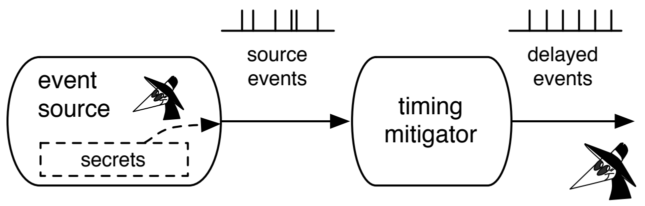

Figure 1: Predictive mitigation

The rest of the paper is structured as follows. Section 2 introduces the extended form of predictive mitigation in the context of the prior work. Section 3 presents different ways to construct predictive mitigators depending on the concurrency model and on what information is considered public. Section 4 analyzes information leakage under various mitigation schemes and assumptions about applications. Section 5 develops formal results on the composition of predictive mitigators. Experiments with applying predictive mitigation to real applications are presented in 6. Related work is discussed in Section 7; the paper concludes in Section 8.

2. Predictive mitigation

Timing channels can be divided into *internal* and *external* timing channels [15]. Predictive timing mitigation is a general method for limiting leakage through *external* channels: those in which the timing measurement is taken external to the system. Because measurement is external, methods that control internal timing channels by preventing effective timing measurement within the system (e.g., [16, 17, 18, 19]) cannot be applied.

Unlike timing mitigation methods that add random delays (e.g., [20, 16]), predictive mitigation bounds the amount of information that leaks through the timing channel, by delaying events according to a schedule that is predicted in advance.

2.1 Background

In the original predictive mitigation work, the system is modeled abstractly as an event source connected to a timing mitigator, as depicted in Figure 1. The timing of events produced by the event source is in general influenced by confidential information. Further, adversary may be able to affect how confidential information influences timing, enabling timing to be used as a covert channel. For example, the adversary might install software onto the event source to modulate the timing of generated events [7].

Events from the event source are delayed by the timing mitigator to reduce the bandwidth of the timing channel. The adversary is assumed to be able to observe the timing of events leaving the mitigator,[1] but can affect the mitigator *only* via the input stream of source events. Generating fake events does not help; the adversary is assumed to be able to identify them.

At any point, the mitigator has a schedule describing when events are supposed to be released. The schedule is a sequence of predictions, each associated with a future point in time. As long as events arrive according to (or ahead of) the schedule, leakage must be low because the number of possible system behaviors observable by the adversary is small.

The event source might fail to behave according to the schedule, in which case the adversary may learn information. The mitigator responds to the misprediction by selecting a new schedule in a way that ensures that total leakage through the timing channel is

[1]The adversary may also be able to partly observe the *contents* of events leaving the mitigator, but this is a storage channel [1], the control of which is orthogonal to the goals of this work.

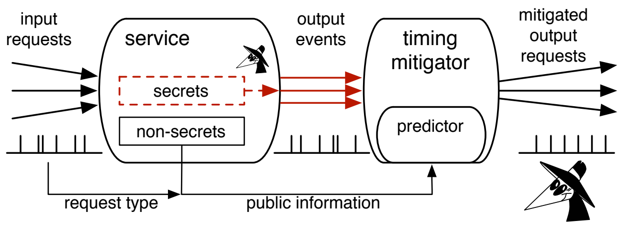

Figure 2: Predictive mitigation of an interactive system

bounded. The period during which the schedule correctly predicts behavior is called an epoch. Schedules are chosen in such a way that the number of epochs grows slowly with time.

For example, consider the following simple "fast doubling" mitigation scheme described by Askarov et al.: Initially, the mitigator has a schedule of predictions at evenly spaced intervals. If the event source fails to deliver events quickly enough, the resulting misprediction causes the mitigator to generate a new schedule in which the interval between predictions is doubled.

We can bound the amount of information that leaks through the adversary's observations through a combinatorial analysis of the number of possible distinct observations the adversary can make. An observation consists of a sequence of times at which events are released by the mitigator. Because events are released in accordance with schedules, the number of possible observations is limited; therefore, the information-theoretic entropy of the timing channel is bounded. This in turn bounds the capacity of the timing channel. In total time T, there can be no more than $\log(T+1)$ epochs,[2] each of which leaks no more than $\log(T+1)+1$ bits of information. Therefore, this simple scheme releases no more than $(1+\epsilon)\log^2 T$ bits of information, where ϵ is small for large T [14]. As this bound shows, it is possible to ensure leakage is asymptotically sublinear over time.

Note that this argument is all about the capacity of the timing channel, without any assumptions about how efficiently secrets are encoded into this channel. The bound applies even if the adversary is perfectly encoding secrets into event timing. But if the adversary does not have this level of control, the bound is likely to be quite conservative.

2.2 Generalizing predictive mitigation

The prior work on predictive mitigation assumes very little about the event source, which means that it can be applied to a wide range of systems. Predictive mitigation can address even difficult low-level timing channels such as those created by hardware contention at the level of the processor or the bus, as long as the mitigator is able to delay externally visible events to precisely the time predicted by the schedule.

However, the very generality of predictive mitigation can make the leakage bounds conservative, and performance of the system is then hurt because the mitigator excessively delays the release of events. By refining the system model, we can make more accurate predictions and also bound timing leakage more accurately. The result is a better tradeoff between security and performance.

Timing channels in network-based services are particularly of interest for timing channel mitigation. These services are interactive systems that accept input requests from a variety of clients and send back responses. Figure 2 illustrates how we extend predictive mitigation for such a system.

Here, the abstract event source used by the prior work is replaced

[2]All logarithms here use base 2.

by a more concrete interactive system that accepts input messages on multiple input channels and delivers output messages to corresponding output channels. Output messages are passed through the timing mitigator, as before, and released by the timing mitigator in accordance with the prediction for that message. If a message arrives early, the mitigator delays it until the predicted time. If it does not arrive in time—a misprediction has happened—the mitigation starts a new epoch and makes a new, more conservative prediction.

This scheme significantly generalizes the original predictive mitigation scheme. First, the time to produce each event is predicted separately, rather than requiring the mitigator to predict the entire schedule in advance—which is rather difficult for an interactive system. Second, the prediction may be computed using any public information in the system. This public information may be anything deemed public (the "non-secrets" in the diagram), possibly including some information about input requests. For example, the mitigator may use the time at which a given input request arrives to predict the time at which the corresponding output will be available for release. The model also permits the content of input requests to be partly public. Each request has an application-defined *request type* capturing what information about the request is public. If no information in the request is public, all requests have the same request type.

To see why this generalizes the original predictive mitigation scheme, consider what happens if the prior history of mitigator predictions is the *only* information considered public when predicting the time of output events. In this case, all predictions within an epoch can be generated at the start of the epoch, yielding a completely determined schedule for the epoch. By contrast, our generalized predictive mitigation can make use of information that was not known at the start of the epoch, such as input time. Therefore, predictions can be made dynamically within an epoch.

2.3 Leakage measures

Two ways to measure information leakage have recently been popular. The information-theoretic measure of *mutual information* has a long history of use; it is advocated, for example, by Denning [21], and has been used for the estimation of covert channel capacity, including timing channel capacity, in much prior work (e.g., [22, 23, 24]). Recently, *min-entropy leakage* has become a popular measure, motivated by the observation that two systems with the same leakage according to mutual information may have very different security properties [25].

Prior work on timing channel mitigation has used one or both of these measures. Fortunately, the style of analysis used here and in prior work on predictive mitigation is sufficiently conservative that it bounds both the mutual information and the min-entropy measures of leakage.

The information-theoretic (Shannon) entropy of a finite distribution X over its n possible values is written as $\mathcal{H}(X)$. It achieves its maximal value of $\log(n)$ bits when all n possible values have equal probability. Suppose that O is the distribution over n possible timing observations by the adversary, and S is the distribution over possible secrets that the adversary wants to learn. The mutual information between O and S, written $\mathcal{I}(O; S)$, is equal to $\mathcal{H}(O) - \mathcal{H}(O|S)$, where $\mathcal{H}(O|S)$ is the conditional entropy of O on S—how much entropy remains in O once S is fixed. In our context, the conditional entropy describes how effectively the adversary encodes the secrets S into the observations O. But since conditional entropy is always positive, the mutual information between O and S is at most $\mathcal{H}(O)$, or $\log(n)$.

Smith argues [25] that the min-entropy of a distribution is a better basis for assessing the vulnerability introduced by quantitative leakage because it describes the chance that an adversary is able to guess the value of the secret in one try. The min-entropy of a distribution is defined as $\mathcal{H}_\infty(O) = -\log V(O)$ where $V(O)$ is the worst-case *vulnerability* of O to being guessed: the maximum over the probabilities of all values in O. Let us write $P(o|s)$ for the probability of observation o given secrets s. Köpf and Smith [13] show that the min-entropy channel capacity from S to O is equal to $\log \sum_{o \in O} \max_{s \in S} P(o|s)$. This capacity is maximized when $P(o|s) = 1$ at all o, in which case it is equal to $\log n$. Therefore $\log n$ is a conservative bound on this measure of leakage as well.

3. Predictions for interactive systems

The system model described in Section 2.2 permits a great deal of flexibility in constructing predictions. We now begin to explore the possibilities.

Throughout the rest of the paper we assume that the mitigator has an internal state, denoted by $\mathbf{St}$. In the simplest schemes, the state only records the number of epochs N, that is, $\mathbf{St} = N$. But more complex internal state is possible, as discussed in Section 4.2.

3.1 Inputs, outputs, and idling

For simplicity, we assume that inputs to and outputs from the interactive system correspond one-to-one: each input has one output and vice versa. If inputs can cause multiple output events, this can be modeled by introducing a schedule for delivering the multiple outputs as a batch.

Many services generate output events only as a response to some external input. In the absence of inputs, such systems are idle and produce no output. If the predictor cannot take this into account when generating predictions, the failure to generate output produces gratuitous mispredictions. With generalized predictive mitigation, these mispredictions can be avoided.

For example, consider applying the original predictive mitigation scheme to a service that reliably generates results in 10ms. If the service is idle for an hour, the series of ensuing mispredictions will inflate the interval between predicted outputs to more than an hour, slowing the underlying service by more than five orders of magnitude. Clearly this is not acceptable.

Consider inputs arriving at times $inp_1, inp_2, \ldots inp_n, \ldots$, where each inp_i is the time of input i. We assume that the mitigator has some public state $\mathbf{St}$, and that this state always includes the index of the current mitigation epoch, denoted by N. Let the prediction for events for state $\mathbf{St}$ be described by a function $p(\mathbf{St})$, where p gives a bound on how long it is expected to take to compute an answer to a request in state $\mathbf{St}$.

Whenever the structure of the mitigator state is understood, we use more concrete notation. For example, in the simple mitigator we have $\mathbf{St} = N$, so we we write $p(N)$ for $p(\mathbf{St})$. Simple fast doubling has the prediction function $p(N) = 2^{N-1}$. For more complex predictors, p might depend on other (public) parameters as well. If $S_N(0)$ is the time of the start of the N-th epoch, subsequent event i in epoch N is predicted to occur at time $S_N(i)$:

$$S_N(i) = \max(inp_i, S_N(i-1)) + p(N)$$

The two terms in the above expression correspond to the predicted start of the computation for event i and the predicted amount of time it takes to compute the output, respectively. To predict the start of computation for event i, we take the later of two times: the time input i is available, and the time event $i-1$ is delivered.

3.2 Multiple input and output channels

Now let us consider mitigation on multiple channels, where requests on different channels may be handled in parallel.

There are at least two reasonable concurrency models. The first model assumes that every request type has an associated process and that processes handling requests of one type do not respond to requests of other types. The second model assumes a shared pool of worker processes that can handle requests of any type as they become available.

In either model, the mitigator is permitted to use some information about which channel an input request arrives on and about the content of the request. This information about the channel and the request is considered abstractly to be the request type of the request. There is a finite set of request types numbered $1, \ldots, R$. Requests coming at time inp with request type r are represented as a pair (inp, r). A *request history* is a sequence of requests $(inp_1, r_1) \ldots (inp_i, r_i) \ldots$, where inp_i is the time of request i, and r_i is the type of the request: $1 \leq r_i \leq R$.

The mitigator makes predictions separately for each request type; however, with multiple request types, an epoch is a period of time during which predictions are met for *all* request types. A misprediction for one request type causes an epoch transition for the mitigator, and may change predictions for every request type. We denote the prediction for computation when mitigator is in state $\mathbf{St}$ on request type r by a function $p(\mathbf{St}, r)$. When the state consists only of the number of epochs ($\mathbf{St} = N$), we simply write $p(N, r)$.

3.2.1 Individual processes per request type

In the case where each request type has its own individual process, the prediction for output event i is

$$S_N(i) = \max(inp_i, S_N(j)) + p(N, r_i)$$

where j is the index of the previous request of type r_i; that is, $j = \max\{j' \mid j' < i \wedge r_i = r_{j'}\}$. Hence $S_N(j)$ is the prediction of the previous request of type r_i. We define $S_N(j)$ to be zero when there are no previous requests of the same type.

Example. Consider a simple system with two request types A and B (for clarity we index request types with letters), and consider a mitigator with these prediction functions $p(N, r)$ for $N = 1$:

N	$p(N, A)$	$p(N, B)$
1	10	100

Assume the following input history: $(2, A)$, $(4, B)$, $(6, A)$, and $(30, B)$. That is, two inputs of type A arrive at times 2 and 6, and two of type B arrive at times 4 and 30.

The inputs $(2, A)$ and $(4, B)$ are the first requests of the corresponding types. The predictions for these requests are

$$S_1(1) = \max(2, 0) + 10 = 12$$
$$S_1(2) = \max(4, 0) + 100 = 104$$

For the next request of type A, the prediction is

$$S_1(3) = \max(6, 12) + 10 = 22$$

This prediction takes into account the amount of time it would take for the process for request type A to finish processing the last input and then to delay the message for $p(1, A)$. Similarly, the predicted output time for the fourth request $(30, B)$ is

$$S_1(4) = \max(30, 104) + 100 = 204$$

3.2.2 Shared worker pool

For a shared pool of worker processes, predictions must be derived more carefully. Suppose the system has at least n worker processes that handle input requests. To compute a prediction for input request i that arrives at time inp_i with type r_i, the mitigator needs to know two terms: when the handling of that request will start, and an estimate of how long it takes to complete the request. We assume that the completion estimate is given by $p(N, r)$ and focus instead on the first term. The main challenge is to predict when a worker will be available to process a request. For this we introduce a notion of *worker predictions*. Intuitively, worker predictions are a data structure internal to the mitigator that allows it to predict when different requests will be picked up by worker processes.

Concretely, worker predictions are n sets $W_1, \ldots, W_n$ in which every W_m contains pairs of the form (i, q). When $(i, q) \in W_m$, it means request i is predicted to be delivered at time q by worker m. Therefore, a given index i appears in at most one of the sets W_m. The function $\mathsf{avail}(W)$ predicts when a worker described by set W will be available, by choosing the time when the worker should deliver its last message.

$$\mathsf{avail}(W) \triangleq \begin{cases} \max\{q \mid (i, q) \in W\} & \text{if } W \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

We describe next the algorithm for computing worker predictions.

Initialization. In the initial state of worker predictions, all sets W_m (for $1 \leq m \leq n$) are empty.

Prediction. Given an event i with input time inp_i and request type r_i, the prediction $S_N(i)$ is computed as follows:

1. The earliest available worker j is predicted to handle request i. Therefore, we find j such that $\mathsf{avail}(W_j) = \min_{1 \leq m \leq n}\{\mathsf{avail}(W_m)\}$

2. Since worker j is assumed to handle request i, we make the following prediction q for the i-th output:

$$q = \max(inp_i, \mathsf{avail}(W_j)) + p(N, r_i)$$

The prediction for S_N is $S_N(i) = q$.

3. Finally, worker predictions are updated with prediction (i, q):

$$W_j := W_j \cup \{(i, q)\}$$

Misprediction. When a misprediction occurs, the mitigator resets the state of worker predictions. Consider a misprediction at time τ_N, which defines the start time of epoch N. We reset the state of worker predictions as follows:

1. For every worker m, we find the earliest undelivered message i' that has been received before the misprediction:

$$i' = \min\{i \mid (i, q) \in W_m \wedge inp_i < \tau_N \leq q\}$$

2. If such i' cannot be found, that is, the set in the previous equation is empty, we set W_m to $\emptyset$. Otherwise, we let $q' = \tau_N + p(N, r_{i'})$ and set $W_m = \{(i', q')\}$.

3. Note that the above step resets the state of each W_m in the worker predictions. Using these reinitialized states, we can compute predictions for the unhandled requests, i.e., all requests j with predicted time q such that $q > \tau_N$ according to the steps 1) and 2) described in *Prediction*.

An example using shared worker pool is presented in the Appendix.

4. Leakage analysis

As in [14], we can use a combinatorial analysis to bound how much information leaks via predictive mitigation in interactive systems. One difference is that we take into account the interactive nature of our model and derive bounds based on the number of input requests and the elapsed time. To conservatively estimate leakage we bound the number of possible timing variations that an adversary can observe, as a function of the running time T and the length of the input history M. Per Section 2.3, the leakage is at most the log of the number of possible observations.

We show that a leakage bound of $O(\log T \times \log M)$ can be attained, with a constant factor that depends on the choice of *penalty policy*. When there is a worst-case execution time for every request, a tighter bound of $O(\log M)$ can be derived.

4.1 Bounding the number of variations

To bound the number of possible timing variations, we need to know three values: (1) the number of timing variations within each epoch, (2) the number of variations introduced by schedule selector, and (3) the number of epochs.

Let us consider the number of variations within each epoch. Because messages within a single epoch are delivered according to predictions, the only source of variations within an individual epoch is *whether* there is a misprediction, and if so, *when* the misprediction occurs. This can be specified by the *length* of the epoch. When the mitigator has received at most M messages, the length of any single epoch can be at most $M + 1$.

When the mitigator transitions from epoch N to epoch $N + 1$, it chooses the schedule for the next epoch. Since the predictor can rely on public information, the "schedule" is actually an algorithm parameterized by public inputs. However, this algorithm may be chosen based on non-public inputs, in which case the choice of schedule may convey additional information to the adversary. Following [14], we denote by Λ_N the number of possible schedules when transitioning between epochs N and $N + 1$. Its value depends on the details of the schedule selector. For simple mitigation schemes, where the choice of the next schedule does not depend on secrets, we have $\Lambda_N = 1$. For *adaptive* mitigation [14], where the choice of schedule depends on internal state such as the size of the mitigator's message buffer, Λ_N may be greater than one.

Consider a mitigator that at time T has received at most M requests and reached at most N epochs. The number of possible timing variations of such a mitigator is at most

$$(M+1)^N \cdot \Lambda_1 \dots \Lambda_N$$

Measured in bits, the corresponding bound on leakage is the logarithm of the number of variations:

$$N \cdot \log(M+1) + \sum_{i=1}^{N} \log \Lambda_i$$

Note that for the simple doubling scheme, because $\Lambda_i = 1$, we also have $\sum_{i=1}^{N} \log \Lambda_i = 0$.

We can enforce an arbitrary *enforcing bound* on leakage. Denote by $B(T, M)$ the amount of information permitted to be leaked by the mitigator. Enforcing bound $B(T, M)$ is satisfied if the mitigator ensures this inequality holds:

$$N \cdot \log(M+1) + \sum_{i=1}^{N} \log \Lambda_i \leq B(T, M)$$

This equation requires a relationship between the number of epochs, the elapsed time, and the number of received messages. The exact nature of this relationship is determined by *penalty policies*.

4.2 Penalty policies

Recall that the function $p(\mathbf{St}, r)$ predicts a bound on computation time for request type r in state $\mathbf{St}$. The intuition is that the more mispredictions have happened in the past (as recorded in $\mathbf{St}$), the larger is the value of $p(\mathbf{St}, r)$. The computation is *penalized* by delivering its response later.

Designing a penalty policy function opens up a space of possibilities. The question is how mispredictions on different request types are interconnected—for example, whether a particular request type should be penalized for mispredictions on other request types, and if so, then how much.

On one side of the spectrum, we can use a *global penalty policy* that penalizes *all* request types when a misprediction occurs. If all request types are penalized, it becomes harder to trigger mispredictions on any of them in future. Therefore, this policy provides a tight bound on N. Intuitively, an adversary gains no additional power to leak information by switching between request types. However, performance of all request types is hurt by mispredictions on any request type.

On the other end of the spectrum is a *local penalty policy* in which request types are not penalized by mispredictions on other types. This improves performance but offers weaker bounds on leakage. To see this, assume that the number of transitions a single request type can make is N. Since penalties are not shared between request types, with R types, as many as $R \times N$ mispredictions can occur. Timing leakage might be high if R is large; intuitively, the adversary can attack each request type independently.

Aiming for more control of the tradeoff between security and performance, we explore penalty policies that fill in the space between the global and local penalty policies. The key insight is that the request types with few mispredictions contribute little to total leakage, so they should share little penalty. This insight brings an l-level *grace period* policy. In a l-level grace period policy, request type r is only penalized by other types when the number mispredictions on r is greater than l.

For more complex penalty policies, leakage analysis becomes more challenging. In Section 4.4, we present an efficient and precise way of bounding N for some penalty policies.

4.3 Generalized penalty policies

Let us refine the state $\mathbf{St}$ to record the number of mispredictions for each request type. If m_r denotes the number of mispredictions on request type r, the mitigator state contains a vector of mispredictions counts $\vec{m} = m_1, \dots, m_R$. Initially all m_r are zero. When a misprediction happens on request type r, vector entry m_r is increased by one. In the following, we assume $\mathbf{St} = \vec{m}$, and write the penalty function as $p(\vec{m}, r)$.

Recall that during an epoch, predictions for all types are met. Given a vector of mispredictions $\vec{m}$, the number of epochs N is simply $N = 1 + \sum_{i=1}^{R} m_i$. Thus, the problem of bounding N is the same as bounding the sum $\sum_{i=1}^{R} m_i$.

For convenience, let us focus on a family of penalty functions p that are a composition of three functions:

$$p(\vec{m}, r) = q(r) \times (\phi \circ \mathrm{idx})(\vec{m}, r)$$

Here function $\phi(n)$ is a *baseline penalty* function, which given a penalty index n returns the prediction for n. The penalty index represents how severely this request type is penalized. It is computed by function $\mathrm{idx}(\vec{m}, r)$, which returns the value of the index in the current state $\vec{m}$ for request type r. Finally, $q(r)$ returns an initial penalty for request type r, and allows us to model different initial estimates of how long it takes to respond to the request of type r. For instance, if one knows that request type r_1 needs at least one second, and request type r_2 needs at least 100 seconds, then one can set $q(r_1) = 1, q(r_2) = 100$.

Examples. For penalty policies based on fast doubling, we set $\phi(n) = 2^n$, and $q(r) = q_0$ for all r with some initial quantum q_0. To use the global penalty policy, idx can be set to $\mathrm{idx}(\vec{m}, r) = \sum_{i=1}^{R} m_i$. To use the local penalty policy, idx is chosen as $\mathrm{idx}(\vec{m}, r) = m_r$. For an l-level grace period policy, we define idx to depend on the parameter l:

$$\mathrm{idx}(\vec{m}, r) = \begin{cases} m_r & \text{if } m_r \le l \\ \sum_{i=1}^{R} m_i & \text{otherwise} \end{cases}$$

4.4 Generalized leakage analysis

As discussed earlier, different penalty functions yield different bounds on N. While it is possible to analyze such bounds for specific penalty policies, in general it is hard to bound leakage for more complex penalty policies.

This section describes a precise method for deriving such bounds for several classes of penalty policies. We transform the problem of finding a bound on the number of epochs N into an optimization problem with R constraints, where R is the number of request types. These constraints can be nonlinear in general, but all considered classes of penalty functions can be solved in constant time.

We focus on penalty functions where $p(\vec{m}, r)$ is monotonic. Because monotonicity is natural for a "penalty", this requirement does not really constrain the generality of the analysis.

State validity. We write $\vec{0}$ for the initial state $\vec{0}$ in which no mispredictions have happened. At the core of our analysis are two notions: state reachability and state validity. Informally, a state $\vec{m}$ is reachable at time T if there is a sequence of mispredictions that, starting from $\vec{0}$, lead to $\vec{m}$ by time T. To bound the number of possible epochs N at time T, it is sufficient to explore the set of all reachable states, looking for $\vec{m}$ in which $1 + \sum m_i$ (and therefore N) is maximized.

Enumerating all reachable states may be infeasible. In particular, an exact enumeration requires detailed assumptions about the thread model presented in Section 3.2. Instead, we overapproximate the set of reachable states for efficient searching of the resulting larger space.

For this, we define the notion of state validity at time T. State validity at time T is similar to reachability at time T, except that we focus only on the predicted time to respond to a request, ignoring the time needed to execute earlier requests.

We first introduce the notion of a valid successor:

DEFINITION 1 (VALID SUCCESSOR). *A state $\vec{m}'$ is a* valid successor *of type j ($1 \le j \le R$) for state $\vec{m}$ when $m'_j = m_j + 1$ and $m'_i = m_i$ for $i \neq j$.*

For example, with three different request types ($R = 3$), the state $(0, 0, 1)$ is a valid successor of type 3 for state $\vec{0}$.

We can then define state validity:

DEFINITION 2 (STATE VALIDITY FOR TIME T). *For penalty function $p(\vec{m}, r)$, a state $\vec{m}$ is a* valid state for time T *if there exists a sequence of request types $j_1, \dots j_{n-1}, j_n$, such that, if $m_0 = \vec{0}$, it holds that for all i, $1 \le i \le n$ we have*

- *$\vec{m}_i$ is valid successor of type j_i for state $\vec{m}_{i-1}$.*
- *$p(\vec{m}_{i-1}, r_{j_i}) \le T$*
- *$\vec{m}_n = \vec{m}$*

The second condition approximates whether the state $\vec{m}_{i-1}$ can make one more transition: if execution time is predicted to exceed T, no more transitions are possible.

Example. Consider the simple case of one request type and time 6 with prediction function $p(\vec{m}, r) = 2^{m_r}$.

State $\vec{m} = (3)$ is a *valid* state for time 6. Consider the request type sequence $1, 1, 1$. We have $\vec{m}_0 = \vec{0}$. Since $\vec{m}_1$ is a valid successor of type 1 for state $\vec{m}_0$, we have $\vec{m}_1 = (1)$. Similarly, we have $\vec{m}_2 = (2)$ and $\vec{m}_3 = (3)$. It is easy to check that $p(\vec{m}_0) = 1 \le 6$, $p(\vec{m}_1) = 2 \le 6$ and $p(\vec{m}_2) = 4 \le 6$. Since $\vec{m}_3 = \vec{m}$, $\vec{m}$ is valid by definition.

However, state $\vec{m}' = (4)$ is not *valid*. Otherwise, since there is only one request type in this example, j_n must be 1. Therefore, $\vec{m}_{n-1}$ must be (3) because $\vec{m}_n$ is a valid successor of type 1 for $\vec{m}_{n-1}$. However, $p(\vec{m}_{n-1}) = 8 > 6$. This contracts the definition of validity.

4.4.1 Transforming to an optimization problem

In this part, we show how to get the maximal $\sum_{i=1}^{R} m_i$ among all valid states when prediction function $p(\vec{m}, r)$ is monotonic. First, we show a useful lemma, proved in the appendix.

LEMMA 1. *Assume $p(\vec{m}, r)$ is monotonic. If $\vec{m}$ is a valid successor of some type j for $\vec{m}'$ such that $p(\vec{m}', j) \le T$, then*

$$\vec{m} = (m_1, \dots, m_R) \text{ is valid for } T \iff$$
$$\vec{m}'' = (m_1, \dots, m_{j-1}, 0, m_{j+1}, \dots, m_R) \text{ is valid for } T$$

Lemma 1 allows us to describe valid states by R constraints. To see this, first observe that because $\vec{m}$ is valid for T, there are some j_1 and $\vec{m}'$ such that $\vec{m}$ is a valid successor of $\vec{m}'$ of type j_1. By Definition 1, $p(\vec{m}', j_1) \le T$. This is our first constraint on the space of valid states.

By Lemma 1, the validity of $\vec{m}$ for T implies the validity of $(m_1, ..., m_{j_1-1}, 0, \dots, m_R)$ for T. Repeating the previous step, there is some $j_2 \neq j_1$ and $\vec{m}''$ where $(m_1, ..., m_{j_1-1}, 0, \dots, m_R)$ is a valid successor of $\vec{m}''$ of type j_2; this gives us the second constraint, $p(\vec{m}'', j_2) \le T$. Proceeding as above, we obtain R constraints such that $\vec{m}$ is valid iff all constraints are satisfied.

Based on the properties of p, our analysis proceeds as follows. We present two different classes of p in the order of difficulty of analyzing them, starting from the easiest.

Symmetric predictions. We first look at prediction policies in which all request types are penalized *symmetrically*:

1. for all i, j, such that $1 \le i, j \le R$ it holds that $p(m_1, \dots, m_i, \dots, m_j, \dots m_R, i) = p(m_1, \dots, m_j, \dots, m_i, \dots m_R, j)$.
2. for all i, j, k, such that $1 \le i, j, k \le R$, where $i \neq k$, and $j \neq k$ it holds that $p(m_1, \dots, m_i, \dots m_j, \dots m_R, k) = p(m_1, \dots, m_j, \dots m_i, \dots m_R, k)$.

These properties allow us to reorder the request types in R constraints that we have obtained earlier. For example, the first of the obtained constraints can be rewritten as $p((m_{j_1} - 1, \dots, m_R), 1) \le T$. Moreover, this allows us to rename the variables in the constraints without loss of generality:

$$\begin{cases} p((m_1 - 1, m_2, \dots, m_R), 1) & \le T \\ p((0, m_2 - 1, \dots, m_R), 2) & \le T \\ \dots & \\ p((0, 0, \dots, m_R - 1), R) & \le T \end{cases}$$

Thus, bounding N is equivalent to finding the maximum sum $\sum_{i=1}^{R} m_i$ satisfying all the conditions.

Examples. It is easy to verify that starting with same initial quantum, global, local, and l-level grace period policies penalize all request types symmetrically. We proceed with the analysis of these policies below.

1. Consider the global penalty function with fast doubling and the starting quantum $q_0 = 1$. The j-th constraint in the above system has form

$$2^{(\sum_{i=j}^{R} m_i - 1)} \le T$$

Here, $N = 1 + \sum_{i=1}^{R} m_i \le \log T + 2$. This is very close to the bound $\log(T+1)+1$ given in [14].[3]

Using the leakage bound derived in Section 4.4, we obtain that for global penalty policy, when the mitigator runs for at most time T the leakage is bounded by function $B(T, M)$ where

$$B(T, M) = (\log T + 2) \cdot \log(M + 1)$$

2. Now consider the local penalty policy with the same penalty scheme and initial quantum. We have R constraints of the form:

$$2^{m_i - 1} \le T, 1 \le i \le R$$

It is easy to derive $N \le R \cdot (\log T + 1) + 1$.

Using this bound for N, we obtain that when the mitigator runs for at most time T, the leakage is bounded by function $B(T, M, R)$ such that

$$B(T, M, R) = (R \cdot (\log T + 1) + 1) \cdot \log(M + 1)$$

3. We revisit the l-level grace period policy last. In this case, the j-th constraint can be split into two cases:

$$\begin{cases} m_j - 1 \le \log T & \text{when } m_j - 1 \le l \\ \sum_{i=j+1}^{R} m_i - 1 \le \log T & \text{when } m_j - 1 > l \end{cases}$$

In general, l is ordinarily smaller than $\log T$, so N is maximized when $m_i = l + 1, 1 \le i \le R - 1$ and $m_R = \lfloor \log T \rfloor + 1$. Thus, $N \le (R - 1) \cdot (l + 1) + \log T + 2$.

Using this bound for N we obtain than when the mitigator runs for at most time T the leakage is bounded by function $B(T, M, R, l)$ such that

$$B(T, M, R, l) = \log(M + 1) \cdot ((R - 1) \cdot (l + 1) + \log T + 2)$$

Non-symmetric predictions. For other types of penalty functions, we can still try to partition request types into subsets such in each subset, request types are penalized symmetrically. We then generate constraints for validity of subsets.

More formally, we say a vector of mispredictions $\vec{m}'$ is a *subvector* of $\vec{m}$ if and only if $m'_i = 0 \lor m'_i = m_i, 1 \le i \le R$. A set of vectors $\vec{m}^1, \ldots, \vec{m}^k$ is a *partition* of $\vec{m}$ if all vectors are subvectors of $\vec{m}$ and for all m_i, there is one and only one $\vec{m}^j$ such that $m_i^j = m_i$.

The following lemma shows that the condition that $\vec{m}$ is valid is stronger than the validity of all subvectors. Thus, the constraints on vectors in a partition overapproximates that on the validity of $\vec{m}$.

LEMMA 2. *When $p(\vec{m}, r)$ is monotonic, $\vec{m}$ is valid at time T $\Longrightarrow$ any subvector of $\vec{m}$ is valid at time T.*

Since there are R non-zero mispredictions among all vectors in the partition, this estimation still gives R constraints.

4.5 Security vs. performance

As discussed informally earlier, the global penalty policy enforces the best leakage bound but has bad performance; the local penalty policy has the best performance but more leakage. We explore this trade-off between security and performance through simulations.

Simulation setup. We simulated a set of interactive system services whose distribution of execution time have various means and variance. The mean is used for the initial penalty. The fast doubling scheme is used, so the prediction function is

$$p(\vec{m}, r) = q(r) \times 2^{\text{idx}(\vec{m}, r)}$$

where $q(r)$ is the mean time of simulated type r. The form of $\text{idx}(\vec{m}, r)$ is defined by penalty policies.

To see the performance for request with different variance, we simulated both *regular types* and *irregular types*. For regular types, the simulated execution time follows Poisson distribution with different means since page view requests to a web page can be modeled as a Poisson process; for irregular types, execution time follows a perturbed normal distribution which avoids negative execution time. The standard deviation is slightly smaller than the mean multiplied by a factor ranging from 2^3 to 2^7, generating around 3 to 7 mispredictions.

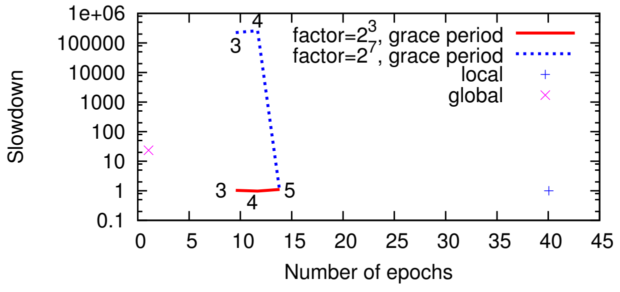

Figure 3: Performance vs. security

Result. The x-axis in Figure 3 shows the bound on number of epochs N and the y-axis shows the normalized slowdown of all simulated request types. All values shown are normalized so that for slowdown, the local policy has value 1 and for the number of epochs, the global policy has value 1.

Results when standard deviation of irregular types has a factor of 2^3 and 2^7 are shown in the figure, which demonstrates the impact of execution-time variation on performance. Number on lines denotes grace-period level.

The results confirm the intuition that the global penalty policy has the best security but bad performance, and the local policy has the best performance. However, we can see the l-level grace period policies have considerably fewer epochs N, yet performance similar to that of the local policy when l is larger than m_{r_i} for most types.

When the variance of execution time increases, small grace-period level ($l = 3, 4$) can bring slowdown that is orders of magnitude higher than in the global case. The reason is that each irregular request type can trigger l mispredictions. Once misprediction of a request type is larger than l, $\text{idx}(\vec{m}, r)$ returns a large number. However using a larger grace-period level ($l = 5$) could restore performance at the cost of more leakage.

Penalty policies with other forms are possible to provide more options between the trade-off of security and performances. We leave a more comprehensive analysis of more penalty policies as future work.

4.6 Leakage with a worst-case execution time

In the analysis above, no assumption is made about execution time for each request type. The adversary can delay responses for an arbitrarily long time to covertly convey more information.

However, for some specific platforms, such as real-time systems and web applications with a timeout setting, we can assume a

[3]Though [14] does not consider request types, the penalty policies considered there are effectively global penalty policies.

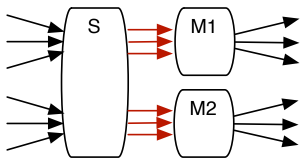

Figure 4: Parallel composition of mitigators

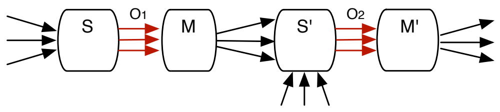

Figure 5: Sequential composition of mitigators

worst-case execution time T_w. Given this constraint, we can derive a tighter leakage bound.

The analysis works similarly to that in Section 4.3, but instead of using the conservative constraint $p(\vec{m}_{i-1}, r_{j_i}) \leq T$ as in Definition 2, worst-case execution time provides a tighter estimation:

$$p(\vec{m}_{i-1}, r_{j_i}) \leq T_w$$

Compared with bounding running time T, this condition more precisely approximates whether the state $\vec{m}_{i-1}$ can make one more misprediction to $\vec{m}_i$. The reason is that whenever $p(\vec{m}_{i-1}, r_{j_i}) > T_w$, the state $\vec{m}_{i-1}$ cannot have another misprediction because execution is bounded by T_w. Therefore, we can reuse the bound on the number of epochs in Section 4.3 by replacing T with T_w.

For example, total leakage with the assumption of worst-case execution time T_w for the global penalty policy is bounded by

$$B(T, M) = (\log T_w + 2) \cdot \log(M + 1)$$

This logarithmic bound is asymptotically the same as that achieved by the less general bucketing scheme proposed by Köpf et al. [12] for cryptographic timing channels.

For the l-grace-period penalty policy we can perform a similar analysis to derive a bound on leakage:

$$B(T, M, R, l) = \log(M + 1) \cdot ((R - 1) \cdot (l + 1) + \log T_w + 2)$$

5. Composing mitigators

If timing mitigation is used, we can expect large systems to be built by composing mitigated subsystems. Askarov et al. [14] show empirically that composing mitigators sequentially performs well, which makes sense because mitigated output has more predictable timing. However, the prior work did not analyze leakage.

We analyze composed mitigators by considering the leakage of two gadgets: two mitigators connected either in parallel or sequentially (Figures 4 and 5). More complex systems with mitigated subsystems can be analyzed by decomposing them into these gadgets.

Parallel composition. Figure 4 is an example of parallel composition of mitigators, in which requests received by the system are handled by two independent mitigators. The bound on the leakage of the parallel composition is no greater than the sum of the bounds of the independent mitigators. To see this, denote by P the total number of variations of the parallel composition, and denote by V_1 and V_2 the number of timing variations of the first and second mitigators, respectively. We know $P \leq V_1 \cdot V_2$; consequently, the total leakage of parallel composition $\log P$ is bounded by $\log V_1 + \log V_2$. The same argument generalizes to n mitigators in parallel.

Sequential composition. Suppose we have a security-critical component, such as an encryption function, and leakage from this component is controlled by a mitigator that guarantees a tight bound, say at most 10 bits of the encryption key. We can show that once mitigated, leakage of the encryption key can never exceed 10 bits, no matter how output of that component is used in the system. This is true for both Shannon-entropy and min-entropy definitions of leakage.

Consider sequential composition of two systems as depicted in Figure 5. Suppose that the secrets in the first system are S, and that the outputs of the first and the second mitigators are O_1 and O_2 respectively. We consider how much the output of each of the mitigators leaks about S.

As discussed in Section 2.3, the leakage of the first mitigator using mutual information is $\mathcal{I}(S; O_1)$ and the leakage of the second is $\mathcal{I}(S; O_2)$. Then we can show that the second mitigator leaks no more information about S_1 than the first does. We formalize this in the following lemma.

LEMMA 3. $\mathcal{I}(S; O_1) \geq \mathcal{I}(S; O_2)$

A similar result holds for min-entropy leakage.

LEMMA 4. $V(S|O_1) \geq V(S|O_2)$

Both of these lemmas are proved in the appendix.

Discussion. Parallel and sequential composition results enable deriving conservative bounds for networks of composed subsystems. The bounds derived may be quite conservative in the case where parallel mitigated systems have no secrets of their own to leak. If the graph of subsystems contains cycles, it cannot be decomposed into these two gadgets. We leave a more comprehensive analysis of mitigator composition to future work.

6. Experiments

To evaluate the performance and information leakage of generalized timing mitigation, we implemented mitigators for different applications. The widely used Apache Tomcat web container was modified to mitigate a local hosted application. We also developed a mitigating web proxy to estimate the overhead of mitigating real-world applications—a non-trivial homepage that results in 49 different requests and a HTTPS webmail service that requires stronger security.

We explored how to tune this general mechanism for different security and performance requirements. The results show that mitigation does slow down applications to some extent; we suggest the slowdown is acceptable for some applications.

6.1 Mitigator design and its limitations

We define the system boundary in the following way. Inputs enter the system at the point when Tomcat dispatches requests to the servlet or JSP code. Results returned from this code are considered outputs. Thus, all timing leakage arising during the processing of the servlet and the JSP files is mitigated.

This implementation of mitigation has limitations. Because of shared hardware and operating-system resources such as filesystem caches, memory caches, buses, and the network, the time required to deliver an application response may convey information about sensitive application data. Our current implementation strategy, chosen for ease of implementation, prevents fully addressing these timing channels where they affect timing outside the system boundary as defined.

To completely mitigate timing channels, mitigation should be integrated at the operating system and hardware levels. For example, the TCP/IP stack might be extended to support delaying packets until a mitigator-specified time. With such an extension, all timing channels, including low-level interactions via hardware caches and bus contention, would be fully mitigated. Although we leave the design of such a mechanism to future work, we see no reason why a more complete mitigation mechanism would significantly change the performance and security results reported here.

6.2 Mitigator implementation

We implemented the mitigator as a Java library containing 201 lines of Java code, excluding comments and the configuration file. This library provides two functions:

```
Mitigator startMitigation (String requestType);
void      endMitigation   (Mitigator miti);
```

The function `startMitigation` should be invoked when an input is available to the system, passing an application-specific request type identifier. The function `endMitigation` is used by the application when an output is ready, and the mitigator for the related input is required for this interface. Calling `endMitigation` blocks the current thread until the time predicted by the mitigator.

Instead of optimizing for specific applications, we heuristically choose the following parameters for all experiments: 1. *Initial penalty*: the initial penalty for all request types is 50 ms, a delay short enough to be unnoticeable to the user. 2. *Penalty policy*: we use the 5-level grace period policy since it provides good tradeoff between security and performance as shown in 4.5. 3. *Penalty function*: most requests are returned within 250 ms, and the distribution is quite even. We evenly divide the first 5 epochs to make predictions more precise: 50 ms, 100 ms, 150 ms, 200 ms, 250 ms, doubling progressively thereafter. 4. *Worst-case execution time* T_w: We assume worst-case execution time for requests T_w to be 300 seconds. This is consistent with Firefox browser version 3.6.12, which uses this value as a default timeout parameter.

6.3 Leakage revisited

Applying the experiment settings into the formula from Section 4.6 with R request types, the following leakage bound obtains:

$$\begin{aligned} &((R-1)\cdot(l+1)+(\log T_w+2))\cdot\log(M+1) \\ =&((R-1)\cdot 6+(\log 300000+2))\cdot\log(M+1) \\ \leq&(6\cdot R+15)\cdot\log(M+1) \end{aligned}$$

where M is the number of inputs using the simple doubling scheme.

Intuitively, introducing more request types helps make the prediction more precise for each request, because processing time varies for different kinds of requests. However, the leakage bound is proportional to the number of request types. So it is important to find the right tradeoff between latency and security.

6.4 Latency and throughput

To enable the mitigation of unmodified web applications, we modified the open source Java Servlet and JavaServer Pages container Tomcat 6.0.29 using the mitigation library.

Experiment setup. Mitigating Tomcat requires only three lines of Java code: one line generating a request type id from the HTTP request, one line to start the mitigation, and another line to end mitigation after the servlet is finished. We deployed a JSP wiki application, JSPWiki[4], in the mitigating Tomcat server to evaluate how mitigation affects both latency and throughput. Measurements were made using the Apache HTTP server benchmarking tool *ab*.[5]

[4]http://www.jspwiki.org

[5]http://httpd.apache.org/docs/2.0/programs/ab.html

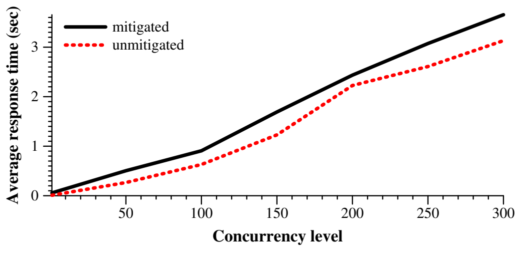

Figure 6: Wiki latency with and without mitigation

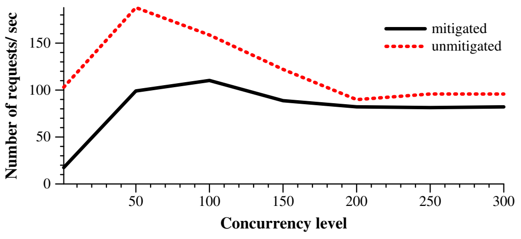

Figure 7: Wiki throughput with and without mitigation

Since we focus on the latency and throughput overhead of requesting the main page of the wiki application, the URI is used as the request type identifier.

Results. We measured the latency and throughput of the main page of JSPWiki for both the mitigated and unmitigated versions. We used a range of different concurrency settings in *ab*, controlling the number of multiple requests to perform at a time. The size of the Tomcat thread pool is 200 threads in the current implementation. For each setting, we measured the throughput for 5 minutes. The results are shown in Figure 6 and Figure 7.

When the concurrency level is 1—the sequential case—the unmitigated Wiki application has a latency around 11ms. Since the initial penalty is selected to be 50ms in our experiments, the average mitigated latency rises to about 57ms: about 400% overhead. This is simply an artifact of the choice of initial penalty.

As we increase the number of concurrent requests, the unmitigated application exhibits more latency, because concurrent requests compete for limited resources. On the other hand, the mitigation system is predicting this delayed time, and we can see that these predictions introduce less overhead: at most 90% after the concurrency level of 50; an even smaller overhead is found for higher concurrency levels.

The throughput with concurrency level 1 is much reduced from the unmitigated case: only about 1/5 of the original throughput. However, when the concurrency level reaches 50, throughput increases significantly in both cases, and the mitigated version has 52.73% of the throughput of the unmitigated version. For higher levels of concurrency, the throughput of the two versions is similar.

6.5 Real-world applications with proxy

We evaluated the latency overhead of predictive mitigation on existing real-world web servers. To avoid the need to deploy predictive mitigation directly on production web servers, we introduce a mitigating proxy between the client browser and the target host. We modified an open source Java HTTP/HTTPS proxy, *LittleProxy*[6], to use the mitigation library, adding about 70 LOC. We

[6]http://www.littleshoot.org/littleproxy/index.html

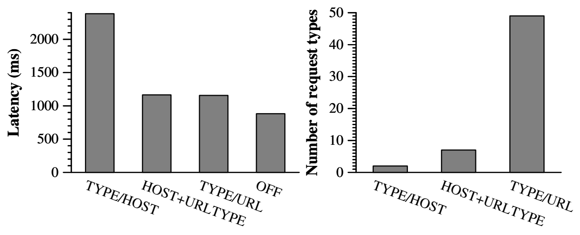

Figure 8: Latency for an HTTP web page

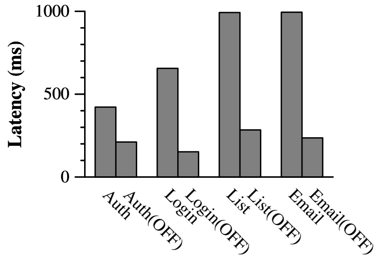

Figure 10: Latency overhead for HTTPS webmail service

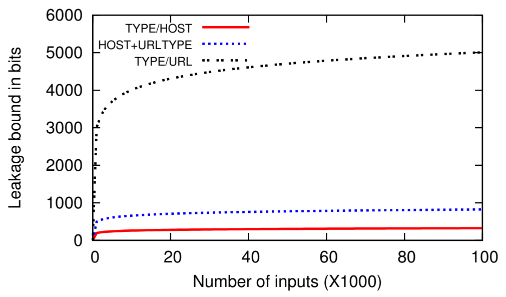

Figure 9: Leakage bound for an HTTP web page

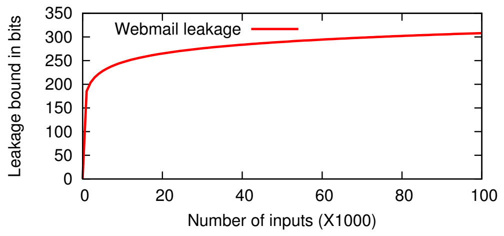

Figure 11: Leakage bound for HTTPS webmail service

used it to evaluate latency with two remote web servers: a HTTP web page and an HTTPS webmail service.

With mitigation again done entirely at user level, timing channels that arise outside the mitigation boundary cannot be mitigated. The mitigation boundary is defined as follows: the mitigating proxy treats requests from client browser as inputs, and forwards these requests to the host. The response from the host is regarded as an output in the black-box model.

The proxy mitigates both the response time of the server and the round-trip time between the proxy and server. Only the first part corresponds to real variation that would occur with a mitigating web server. To estimate this part of latency overhead, we put the proxy in a local network with the real host. Because we found measured little variation in this configuration, the results here should estimate latency for real-world applications reasonably accurately.

6.5.1 HTTP web page

Unlike the previous stress test that requests only one URL, we evaluated latency overhead using a non-trivial HTTP web page, a university home page that causes 49 different requests to the server. Multiple requests bring up the opportunity of tuning the tradeoff between security and performance. Various ways to choose request types were explored:

1. TYPE/HOST: all URLs residing on the same host are treated as one request type, that is, they are predicted the same way.

2. HOST+URLTYPE: different requests on the same host are predicted differently based on the URL type of the request. We distinguish URL types based on the file types, such JPEG files, CSS files and so on. Each of them corresponds to a different request type.

3. TYPE/URL: individual URLs are predicted differently.

Figure 8 shows the latency of loading the whole page along with the number of request types with these options. The results show that in the most restrictive TYPE/HOST case, latency is almost tripled compared to the unmitigated case. HOST+URLTYPE and TYPE/URL options have similar latency results, with around 30% latency overhead.

From the security point of view, the TYPE/HOST option only results in two request types: one host is in the organization, and the other one is `google-analytics.com`, used for the search component in the main page. HOST+URLTYPE introduces 6 more request types, while using the TYPE/URL option, there are as many as 49 request types. The information leakage bounds for different options are shown in Figure 9.

The HOST+URLTYPE choice provides a reasonable tradeoff: it has roughly a 30% latency overhead, yet information leakage is below 850 bits for 100,000 requests.

6.5.2 HTTPS webmail service

We also evaluate the latency with a webmail service based on Windows Exchange Server. After the user passes Kerberos-based authentication (Auth), he is redirected to the login page (Login) and may then see the list of emails (List) or read a message (Email).

Request type selection. This application accesses sensitive data, so we evaluate performance with the most restrictive scheme: one request type per host. There are actually two hosts: one host is used to serve only AuthPage.

Results. We measured the latency overhead of four representative pages for this service. The number of different requests generated by these pages ranges from 6 to 45. The results in Figure 10 show that the latency overhead ranges from 2 times to 4 times for these four pages; in the worst case, latency is still less than 1 second. Also, this overhead can be reduced with different request type selection options.

Figure 11 shows the leakage bound of this mitigated application. The leakage is limited to about 300 bits after 100,000 requests and grows slowly thereafter.

7. Related work

The most closely related work is that of Askarov et al. [14]. Comparisons to that work have been made throughout the paper; at a high level, the generalized predictive mitigation scheme makes possible the practical application of predictive mitigation to general services. The simple predictive mitigator defined by Askarov et al. is manifestly unsuitable to this task, as discussed in Section 3.1.

Köpf et al. [12, 13] introduced the mechanism of *bucketing* to mitigate timing side channels in cryptographic operations, achieving asymptotically logarithmic bounds on information leakage but with stronger assumptions than in this work. Their security analyses rely on the timing behavior of the system agreeing with a previously measured distribution of times; therefore they implicitly assume that the adversary does not control timing, and that there is a worst-case execution time. The bucketing approach does not achieve logarithmic bounds for general computation.

The NRL Pump [26] and its follow-ups, like Network Pump [27], are also network service handling that handle requests. The Pump work addresses timing channels arising from message acknowledgments (which correspond to but are less general than outputs in this work). Acknowledgment timing is stochastically modulated using a moving average of past activity, and leakage in one window does not affect later windows. Therefore the NRL/Network Pumps can enforce only a linear leakage bound.

Much other work has studied timing channels at the network level, exploring techniques such as adding random delays or periodic quantization of time (e.g., [24, 20]). For discussion of this prior work, see [14]. Work on language-based security has also addressed timing channels, especially for internal timing channels, and this also is covered in [14].

8. Conclusion

Predictive mitigation as introduced earlier offered the possibility of mitigating timing channels in general computations, but was impractical as a way to build real networked services. In this work, we have both generalized and refined the original model of predictive mitigation to apply to interactive systems. The experimental results from the implementation of this generalized prediction mitigation scheme suggest that it may be a practical way to mitigate timing channels in a variety of networked services.

Acknowledgments

We thank Owen Arden for useful discussions about this work, and the anonymous reviewers for helpful feedback.

This work was funded by a grant from the Office of Naval Research (ONR N000140910652), and by two grants from the NSF: 0424422 (the TRUST center), and 0964409.

9. References

[1] B. W. Lampson, "A note on the confinement problem," *Comm. of the ACM*, vol. 16, no. 10, pp. 613–615, Oct. 1973.

[2] P. Kocher, "Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems," in *Advances in Cryptology—CRYPTO'96*, Aug. 1996.

[3] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, Jan. 2005.

[4] D. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of AES," *Topics in Cryptology–CT-RSA 2006*, Jan. 2006. [Online]. Available: http://www.springerlink.com/index/F52X1H55G1632L17.pdf

[5] A. Bortz and D. Boneh, "Exposing private information by timing web applications," in *Proc. 16th Int'l World-Wide Web Conf.*, May 2007.

[6] G. Shah, A. Molina, and M. Blaze, "Keyboards and covert channels," *Proc. 15th USENIX Security Symp.*, Aug. 2006.

[7] H. Meer and M. Slaviero, "It's all about the timing..." in *Proc. Black Hat USA*, 2007.

[8] Y. Liu, D. Ghosal, F. Armknecht, A. Sadeghi, and S. Schulz, "Hide and seek in time—robust covert timing channels," in *ESORICS*, 2009.

[9] R. G. Gallagher, "Basic limits on protocol information in data communication networks," *IEEE Transactions on Information Theory*, vol. 22, no. 4, Jul. 1976.

[10] M. Padlipsky, D. Snow, and P. Karger, "Limitations of end-to-end encryption in secure computer networks," Mitre Corp., Tech. Rep. ESD TR-78-158, 1978.

[11] I. S. Moskowitz and M. H. Kang, "Covert channels—here to stay?" in *COMPASS '94*, 1994.

[12] B. Köpf and M. Dürmuth, "A provably secure and efficient countermeasure against timing attacks," in *2009 IEEE Computer Security Foundations*, Jul. 2009.

[13] B. Köpf and G. Smith, "Vulnerability bounds and leakage resilience of blinded cryptography under timing attacks," in *2010 IEEE Computer Security Foundations*, Jul. 2010.

[14] A. Askarov, D. Zhang, and A. C. Myers, "Predictive black-box mitigation of timing channels," in *ACM Conf. on Computer and Communications Security (CCS)*, 2010, pp. 297–307.

[15] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in *Proc. 13th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, Jul. 2000, pp. 200–214.

[16] W.-M. Hu, "Reducing timing channels with fuzzy time," in *IEEE Symposium on Security and Privacy*, 1991, pp. 8 – 20.

[17] J. Agat, "Transforming out timing leaks," in *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, Boston, MA, Jan. 2000, pp. 40–53.

[18] S. Zdancewic and A. C. Myers, "Observational determinism for concurrent program security," in *Proc. 16th IEEE Computer Security Foundations Workshop*, Pacific Grove, California, Jun. 2003, pp. 29–43.

[19] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld, "Closing internal timing channels by transformation," in *Proc. 11th Annual Asian Computing Science Conference (ASIAN)*, 2006.

[20] J. Giles and B. Hajek, "An information-theoretic and game-theoretic study of timing channels." *IEEE Transactions on Information Theory*, vol. 48, no. 9, pp. 2455–2477, 2002.

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

[22] J. K. Millen, "Covert channel capacity," in *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, Apr. 1987.

[23] ——, "Finite-state noiseless covert channels," in *Proc. 2nd IEEE Computer Security Foundations Workshop*, Jun. 1989, pp. 11–14.

[24] I. S. Moskowitz and A. R. Miller, "The channel capacity of a certain noisy timing channel," *IEEE Trans. on Information Theory*, vol. 38, no. 4, pp. 1339–1344.

[25] G. Smith, "On the foundations of quantitative information flow," *Proc. 12th Intl' Conf. on Foundations of Software Science and Computation Structures*, pp. 388–402, 2010.

[26] M. H. Kang and I. S. Moskowitz, "A pump for rapid, reliable, secure communication," in *ACM Conf. on Computer and Communications Security (CCS)*, Nov. 1993, pp. 119–129.

[27] M. H. Kang, I. S. Moskowitz, and D. C. Lee, "A network pump," *IEEE Transactions on Software Engineering*, vol. 22, pp. 329–338, 1996.

[28] T. Cover and J. Thomas, *Elements of information theory*. Wiley, 2006.

APPENDIX

Example of shared worker pool

Reusing the settings from the example in Section 3.2.1, we have four inputs: $(2, A)$, $(4, B)$, $(6, A)$ and $(30, B)$, and prediction function $p(1, A) = 10$ and $p(1, B) = 100$. Suppose we have two shared workers.

As described above, the worker predictions are both initialized to be empty: $W_1 = \emptyset$ and $W_2 = \emptyset$. For the first input, both workers are available; that is, $\mathsf{avail}(W_1) = \mathsf{avail}(W_2) = 0$ since W_1 and W_2 are all empty sets now. We break the tie by selecting the worker with smaller index, worker 1, and then we set the prediction for input $(2, A)$ as

$$S_1(1) = \max(2, 0) + 10 = 12$$

Finally, the worker prediction of worker 1 is updated to $\{(1, 12)\}$.

For the second input, $\mathsf{avail}(W_1) = 12$ and $\mathsf{avail}(W_2) = 0$. Worker 2 is the earliest available worker. Similarly to the first input, the prediction for the second output is $S_1(2) = \max(4, 0) + 100 = 104$. The worker prediction of worker 2 is updated to $\{(2, 104)\}$.

Computation of the predicted worker becomes more interesting for the third input $(6, A)$. We have

$$\begin{aligned}\mathsf{avail}(W_1) &= \max\{q \mid (i, q) \in \{(1, 12)\}\} = 12\\ \mathsf{avail}(W_2) &= \max\{q \mid (i, q) \in \{(2, 104)\}\} = 104\end{aligned}$$

The mitigator picks the worker with earliest availability, worker 1. The third output is predicted at: $S_1(3) = \max(6, 12) + 10 = 22$, and the prediction for worker 1 is updated to $\{(1, 12), (3, 22)\}$.

For the last input $(30, B)$, the mitigator first computes the available times for both workers:

$$\begin{aligned}\mathsf{avail}(W_1) &= \max\{q \mid (i, q) \in \{(1, 12), (3, 22)\}\} = 22\\ \mathsf{avail}(W_2) &= \max\{q \mid (i, q) \in \{(2, 104)\}\} = 104\end{aligned}$$

Based on these values, the mitigator picks worker 1 as the predicted worker for the fourth input. The prediction for corresponding output is $S_1(4) = \max(30, 22) + 100 = 130$, and the prediction of worker 1 becomes $\{(1, 12), (3, 22), (4, 130)\}$.

Proof of Lemma 1

Proof. $\Longleftarrow$: since $\vec{m}''$ is valid, there is sequence of request types where all intermediate states satisfy the constraints. Further, we can construct a sequence of request types from $\vec{m}''$ to $\vec{m}$ by appending j to the previous sequence until $\vec{m}_i = \vec{m}$. Since $p(\vec{m}', j) \leq T$ and p is monotonic, all new states corresponding to this sequence still satisfy the constraints.

$\Longrightarrow$: by definition, there is a sequence of request types $r_1, \ldots, r_n$ such that all intermediate states satisfy constraints. Moreover, there must be a point i in this sequence such that $\forall l < i, r_l \neq j$ and $r_i = j$. Thus, the j-th element of $\vec{m}_{i-1}$ is 0.

Then, a new sequence of request types $p_1, \ldots, p_m$ exists such that $p_l = r_l, 0 \leq l \leq i - 1$. For $l \geq i$, if $r_l = j$, skip this type. Otherwise, add the same type to sequence $\vec{p}$. By this construction, two properties of states occurring with $\vec{p}$ are that the j-th element is always 0, and that there is a corresponding state with sequence $\vec{r}$ such that they only differ in the j-th element. We denote the final states with request type sequence $\vec{r}$, $\vec{p}$ as $\vec{m}^r$ and $\vec{m}^p$ respectively. Since state $\vec{m}^r$ satisfies $p(\vec{m}^r, r_l) \leq T$, by monotonicity, corresponding state $\vec{m}^p$ also satisfies this condition. Since $m_n^r = \vec{m}''$, $\vec{m}''$ is valid at T.

□

Proof of Lemma 2

Proof. By definition, there is a sequence of request types $j_1, \ldots, j_n$ such that all conditions in Definition 2 are satisfied. For any subvector of $\vec{m}$, say $\vec{m}'$, we can take a projection of the sequence so that only the request types nonzero in the subvector are kept.

By monotonicity, it is easy to check that all conditions hold in the definition. Moreover, $\vec{m}_n = \vec{m}'$. So $\vec{m}'$ is valid by definition.

□

Proof of Lemmas 3 and 4

We can view the outputs O_1 and O_2 as discrete random variables. Since the second service and its mitigator do not share secret S, the conditional distribution of O_2 depends only on O_1 and is conditionally independent of S (in other words, random variables S, O_1, O_2 form a Markov chain). Denoting the probability mass function of a discrete random variable X as $P(X)$, the joint distribution of these three random variables has probability mass function $P(s, o_1, o_2) = P(s)P(o_1|s)P(o_2|o_1)$. The marginal distribution $P(o_2, s)$ is $\sum_{o_1 \in O_1} P(s, o_1, o_2)$, and for any o_1, we have $\sum_{o_2 \in O_2} P(o_2|o_1) = 1$.

Proof of Lemma 3.

Proof. The proof follows from the standard *data-processing inequality* [28] and the symmetry of mutual information:

$$\begin{aligned}\mathcal{I}(S; O_2) + \mathcal{I}(S; O_1|O_2) &= \mathcal{I}(S; O_1, O_2)\\ &= \mathcal{I}(S; O_1) + \mathcal{I}(S; O_2|O_1)\end{aligned}$$

Note that S and O_2 are conditionally independent given O_1, since the second mitigator produces outputs based on only the output of the first mitigator M, public inputs, and secrets other than S. Thus $\mathcal{I}(S; O_2|O_1) = 0$. Replacing this term with zero in the above equation, we get

$$\mathcal{I}(S; O_2) + \mathcal{I}(S; O_1|O_2) = \mathcal{I}(S; O_1)$$

Also, we know that $\mathcal{I}(S; O_1|O_2) \geq 0$, so we have

$$\mathcal{I}(S; O_1) \geq \mathcal{I}(S; O_2)$$

□

Proof of Lemma 4. As discussed in Section 2.3, min-entropy channel capacity is defined as the maximal value of $\log \frac{V(S|O)}{V(S)}$ among all distributions on S. So it suffices to show $V(S|O_1) = V(S|O_2)$ for any distribution on S.

$$\begin{aligned}V(S|O_2) &= \sum_{o_2 \in O_2} \max_{s \in S} P(s)P(o_2|s)\\ &= \sum_{o_2 \in O_2} \max_{s \in S} \sum_{o_1 \in O_1} P(s, o_1, o_2)\\ &= \sum_{o_2 \in O_2} \max_{s \in S} \sum_{o_1 \in O_1} P(s)P(o_1|s)P(o_2|o_1)\\ &\leq \sum_{o_2 \in O_2} \sum_{o_1 \in O_1} P(o_2|o_1) \max_{s \in S} P(s)P(o_1|s)\\ &= \sum_{o_1 \in O_1} \max_{s \in S}(P(s)P(o_1|s)) \sum_{o_2 \in O_2} P(o_2|o_1)\\ &= \sum_{o_1 \in O_1} \max_{s \in S} P(s)P(o_1|s)\\ &= V(S|O_1)\end{aligned}$$