Information Leakage in Arbiter Protocols [⋆]

Nestan Tsiskaridze[*], Lucas Bang[†], Joseph McMahan[*], Tevfik Bultan[*], and Timothy Sherwood[*]

[*]University of California, Santa Barbara 93106, USA
{nestan,jmcmahan,bultan,sherwood}@cs.ucsb.edu

[†]Harvey Mudd College, Claremont California 91711, USA
bang@cs.hmc.edu

Abstract. Resource sharing while preserving privacy is an increasingly important problem due to a wide-scale adoption of cloud computing. Under multitenancy, it is common to have multiple mutually distrustful "processes" (e.g. cores, threads, etc.) running on the same system simultaneously. This paper explores a new approach for automatically identifying and quantifying the information leakage in protocols that arbitrate utilization of shared resources between processes. Our approach is based on symbolic execution of arbiter protocols to extract constraints relating adversary observations to victim requests, then using model counting constraint solvers to quantify the information leaked. We present enumerative and optimized methods of exact model counting, and apply our methods to a set of nine different arbiter protocols, quantifying their leakage under different scenarios and allowing for informed comparison.

Keywords: arbiter protocols, quantitative information flow, model counting, symbolic execution

1 Introduction

Many of the computer systems we use today have access to secret information, confidentiality of which should not be compromised. In program analysis, methods of *secure information flow* (SIF) are dedicated to tracking the propagation of sensitive information through a program. SIF methods aim to produce a binary answer: *yes,* there is an information leak, or *no,* there is not, and have seen success in verifying anonymity protocols [13], firewall protocols [3], and network security protocols [7]. However, a binary answer to information leakage is not sufficient in general, due to cost of establishing strict non-interference, side-channels

[⋆] This material is based on research sponsored by DARPA under the agreement number FA8750-15-2-0087 and the National Science Foundation under Grants No. 1740352, 1730309, 1717779, 1563935. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

that may leak information through non-functional properties of a system, or due to application semantics that require some information leakage (for example, a password checker always leaks information by reporting if the input matched the secret password). Hence, the general question about information flow in a computer system is not *if* information leaks, but *how much* information leaks? This "how much" question led to the development of *Quantitative Information Flow* (QIF) techniques, which provide a foundational framework for measuring information leakage [25].

In this paper, we present a QIF technique for assessment and comparison of information leakage among resource sharing protocols. Various arbiter protocols have been developed for coordinating processes that share common resources [11]. An arbiter takes resource requests and grants access to the resource based on its policy. We assume that the requests made by one process should not be revealed to another process. In an ideal situation no process should reveal any information to another process unless it is intentional. In reality, many designs need to leak some degree of information to meet other design goals. We demonstrate that using the QIF technique we present one can determine and compare the *amount* of information leakage for different arbiter protocols.

Previous work on information flow properties of protocols has been limited. The techniques we present in this paper introduce a new dimension in protocol analysis, and provide a new way to classify protocols with respect to the amount of information they leak. Interestingly, as our experiments demonstrate, reducing information leakage can conflict with other desirable properties of protocols. For example, improving resource usage or fairness in a given protocol could increase the amount of information leaked.

Our approach is based on symbolic execution and constraint model counting techniques and can handle ramdomized protocols. Given a protocol specification, we extend symbolic execution to extract constraints characterizing relationships between the secret and the adversary-observable events. With model counting constraint solvers, we quantify the amount of information leaked, in terms of entropy, by observable events. We present a novel, efficient and exact model counting technique for a class of constraints extracted during QIF analysis of arbiter protocols.

The rest of the paper is organized as follows. Section 2 discusses different arbiter protocols to be analyzed; Section 3 explains our method of computing leakage of the protocols. Section 4 contains our optimized method of constraint counting, vastly improving performance of the analysis. Section 5 gives our experimental results, Section 6 discusses related work, and Section 7 concludes.

2 Arbiter Protocols

We model synchronous arbiter protocols as a multi-process, multi-round model with n processes and k rounds. Each process i in each round j sends the arbiter a request bit for a shared resource (R_{ij}, where if the bit is one the process is requesting the resource), and receives a grant bit (G_{ij}, where if the bit is one the request is granted) as a response from the arbiter. The basic arbiter protocol

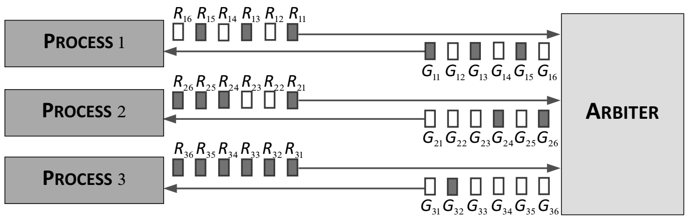
PROCESS 1
PROCESS 2
PROCESS 3
ARBITER
R_{16} R_{15} R_{14} R_{13} R_{12} R_{11}
G_{11} G_{12} G_{13} G_{14} G_{15} G_{16}
R_{26} R_{25} R_{24} R_{23} R_{22} R_{21}
G_{21} G_{22} G_{23} G_{24} G_{25} G_{26}
R_{36} R_{35} R_{34} R_{33} R_{32} R_{31}
G_{31} G_{32} G_{33} G_{34} G_{35} G_{36}

Fig. 1: Arbiter protocol model. Shaded box depicts a bit set to one, white – to zero. Number of processes is 3, rounds 6.

architecture is shown in Figure 1. In the protocols we analyze only one process can be granted access to the shared resource in each round. The basic problem is whether an adversary process can infer the sequence of request bits of another process from the grant bits that adversary receives, and to what extent.

Example. Consider an arbiter protocol that resolves simultaneous requests for the same resource by giving access to the process with the minimum process ID, (e.g. the PRIORITY procedure in Figure 2, also depicted in Figure 1). Suppose an adversary controls Process 2 and targets a victim Process 1. If Process 2 requests access to the resource and does not get the access granted, it is so because the Process 1 has also requested access during the same arbitration round. On the other hand, if Process 2 is granted the access, it must be the case that Process 1 did not request in that round. Consequently, Process 2 can fully infer the request pattern of Process 1. Now suppose the adversary controls Process 3, makes a request, and does not get the access granted. Then the attacker can infer that either Process 1 or Process 2 or both have requested, but cannot distinguish among these cases based on its own response from the arbiter, thereby learning only partial information. In fact, the best strategy for the adversary is to keep requesting in each round, as Process 3 in Figure 1. One expects that resolving resource-request races randomly (e.g. the RANDOM procedure in Figure 2) should not allow one process to infer the request pattern of another process from its own pattern of access grants.

For more complex protocols, it becomes difficult to manually reason about the information flow properties. In this paper, we give automatic techniques for quantifying the amount of information that can be gained from an arbiter protocol by any process about any other process.

We categorize arbiter protocols based on three characteristics: (I) how the concurrent requests are resolved; (II) whether the protocols are stateful or stateless; and (III) whether the processes are stateful or stateless.

We say a protocol (or a process) is *stateless* if access grants (respectively requests) made at each round are independent from those of the previous rounds; and is *stateful* otherwise. Among the stateful process behaviors, we consider the one in which every process holds each initiated request without interruption across the rounds until the arbiter grants access to this process, after which the process is unconstrained on when to initiate a new request.

We resolve concurrent request in three ways: (i) based on a predefined static priority, (ii) based on a dynamically-defined priority, or (iii) randomly. We define a static priority based on the process IDs—the lower a process ID, the higher its priority. Dynamic priorities are defined in, and by, the stateful protocols where the priority of a process at the current round depends on the requests and grants for all processes made at the previous rounds. Concurrences are resolved randomly either in a uniformly-distributed random or a weighted random manner. Handling random components in symbolic analysis is a challenging task on its own. We introduce our approach for extending the quantitative symbolic analysis to support symbolic random components in the arbiter protocols in Section 3.3.

Below, we present various arbiter protocols and discuss how to quantify their information flow properties using our automated approach. We give pseudocode of arbiter protocols for a single round. Let $\boldsymbol{P} = \{P_1, P_2, \ldots, P_n\}$ be a list of processes communicating with an arbiter. In a single round, the arbiter receives a list of requests from these processes $\boldsymbol{R} = \{R_1, R_2, \ldots, R_n\}$ for a shared resource, and returns a grant response to each of the processes, $\boldsymbol{G} = \{G_1, G_2, \ldots, G_n\}$. The requests R_i and grants G_i are modeled to take Boolean values: $\top$ if the request (respectively, grant) is instantiated (respectively, granted), and $\bot$ otherwise.

Stateless arbiters. A *priority-based arbiter* (PRIORITY) and a *randomized arbiter* (RANDOM) are stateless arbiters which differ by how they resolve concurrent requests when multiple processes place a request within the same round.

1) The PRIORITY arbiter resolves concurrent requests based on a predefined *static priority*, always granting access to the process with the highest priority. Without loss of generality, we assume the order $P_1 \succ P_2 \succ \cdots \succ P_n$ on the processes and say that P_1 has the highest priority and P_n the lowest.

2) The RANDOM arbiter resolves concurrent requests randomly.

Pseudocode for a single round of these protocols is shown in Figure 2. ISRACE($\boldsymbol{R}$) routine returns *true* if and only if multiple processes request concurrently. PICKRND($\boldsymbol{R}$) randomly selects a process, among those racing, with equal probability. If a single process requests, FINDREQ($\boldsymbol{R}$) returns the ID of this process, and returns NULL when no process requests.

```
Procedure PRIORITY
Input: $\boldsymbol{R}[1..n]$ an array of requests
Output: $\boldsymbol{G}[1..n]$ an array of responses
1: $\boldsymbol{G} \leftarrow (\bot, \ldots, \bot)$
2: for $i \leftarrow 1$ to $n$ do
3:     if $\boldsymbol{R}[i] = \top$ then
4:         $\boldsymbol{G}[i] \leftarrow \top$
5:         break
6:     end if
7: end for
8: return $\boldsymbol{G}$
```

```
Procedure RANDOM
Input: $\boldsymbol{R}[1..n]$ an array of requests
Output: $\boldsymbol{G}[1..n]$ an array of responses
1: $\boldsymbol{G} \leftarrow (\bot, \ldots, \bot)$
2: if ISRACE($\boldsymbol{R}$) then
3:     $\boldsymbol{G}$[PICKRND($\boldsymbol{R}$)] $\leftarrow \top$
4: else
5:     $pid \leftarrow$ FINDREQ($\boldsymbol{R}$)
6:     if $pid \neq$ NULL then
7:         $\boldsymbol{G}[pid] \leftarrow \top$
8:     end if
9: end if
10: return $\boldsymbol{G}$
```

Fig. 2: Priority and Random Arbiters.

Stateful arbiters. This category includes a *round robin arbiter* (ROUNDROBIN), a *lottery-based arbiter* (LOTTERY), a *first-come-first-serve-based arbiter* (FCFS), and a *longest-idle-based arbiter* (LONGESTIDLE) as shown in Figures 3-5. The concurrences are resolved with a *dynamic priority* order on the processes based on the history of the previous rounds.

3) The RoundRobin arbiter grants access to processes in a circular order by passing around a token incremented at each round: if a process with an ID equal to the value of the token has requested access in a given round the arbiter grants access to this process, otherwise the arbiter does not grant access to any process and moves to the next round with the incremented token. When the token reaches the last process ID it resets to the first one.

4) RoundRobinSkip is a variant of the round robin protocol that never passes a round without a grant when there is a requesting process. The routine FindFirst($\boldsymbol{R}, tkn$) returns an ID of the first requesting process it finds starting from the token and following in a circular manner by skipping over the idle processes that made no request in a given round; if no process made a request in the round—the routine returns NULL.

```
Global: $tkn$
Procedure RoundRobin
Input: $\boldsymbol{R}[1..n]$ an array of requests
Output: $\boldsymbol{G}[1..n]$ an array of responses
1: $\boldsymbol{G} \leftarrow (\bot, \ldots, \bot)$
2: if $tkn = n + 1$ then $tkn \leftarrow 1$
3: end if
4: if $\boldsymbol{R}[tkn]$ then
5:     $\boldsymbol{G}[tkn] \leftarrow \top$
6: end if
7: $tkn \leftarrow tkn + 1$
8: return $\boldsymbol{G}$
```

```
Global: $tkn$
Procedure RoundRobinSkip
Input: $\boldsymbol{R}[1..n]$ an array of requests
Output: $\boldsymbol{G}[1..n]$ an array of responses
1: $\boldsymbol{G} \leftarrow (\bot, \ldots, \bot)$
2: if $tkn = n + 1$ then $tkn \leftarrow 1$
3: end if
4: $pid$ = FindFirst($\boldsymbol{R}, tkn$)
5: if $pid \neq$ NULL then
6:     $\boldsymbol{G}[pid] \leftarrow \top$
7:     $tkn \leftarrow pid + 1$
8: end if
9: return $\boldsymbol{G}$
```

Fig. 3: Round Robin and Round Robin Skip Arbiters.

5) The Lottery arbiter selects a process in a weighted-random manner. In contrast with the Random arbiter, it counts the wait-times of the processes that have been waiting for the access to be granted and resolves concurrent requests by probabilistically prioritizing processes with longer waiting time. $\boldsymbol{W} = (W_1, \ldots, W_n)$ is a list of wait-times of each process. PickRnd($\boldsymbol{W}$) selects a process among the racing ones in a weighted-random manner.

```
Global: $\boldsymbol{W}[1..n]$ an array of wait-times
Procedure Lottery
Input: $\boldsymbol{R}[1..n]$ an array of requests
Output: $\boldsymbol{G}[1..n]$ an array of responses
1: $\boldsymbol{G} \leftarrow (\bot, \ldots, \bot)$
2: for $i \leftarrow 1$ to $n$ do
3:     if $\boldsymbol{R}[i] = \top$ then
4:         $\boldsymbol{W}[i] \leftarrow \boldsymbol{W}[i] + 1$
5:     else
6:         $\boldsymbol{W}[i] \leftarrow 0$
7:     end if
8: end for
9: if IsRace($\boldsymbol{R}$) then
10:     $pid \leftarrow$PickRnd($\boldsymbol{W}$)
11: else
12:     $pid \leftarrow$FindReq($\boldsymbol{R}$)
13: end if
14: if $pid \neq$ NULL then
15:     $\boldsymbol{G}[pid] \leftarrow \top$
16:     $\boldsymbol{W}[pid] \leftarrow 0$
17: end if
18: return $\boldsymbol{G}$
```

Fig. 4: Lottery Arbiter.

6) The FCFS (first-come-first-served) arbiter resolves concurrent requests by considering wait-times of the processes $\boldsymbol{W}$. The AllMax($\boldsymbol{W}$) routine returns the IDs of the processes with the maximal wait-time. If multiple processes have been waiting for the permission grant for the same number of rounds, PickOne() breaks ties. We consider two approaches for PickOne(): based on the static priority where the process with the lowest ID gets access, and uniformly random.

7) The LongestIdle arbiter does the opposite to the FCFS in the sense that it prioritizes processes by length of idle time. $\boldsymbol{I} = (I_1, \ldots, I_n)$ is a list of idle-times of each process. Ties are broken in the same manner as in FCFS.

```
Global: $\boldsymbol{W}[1..n]$ an array of wait-times
Procedure FCFS
Input: $\boldsymbol{R}[1..n]$ an array of requests
Output: $\boldsymbol{G}[1..n]$ an array of responses
$\boldsymbol{G} \leftarrow (\bot, \ldots, \bot)$
for $i \leftarrow 1$ to $n$ do
    if $\boldsymbol{R}[i] = \top$ then
        $\boldsymbol{W}[i] \leftarrow \boldsymbol{W}[i] + 1$
    else
        $\boldsymbol{W}[i] \leftarrow 0$
    end if
end for
if IsRace($\boldsymbol{R}$) then
    $pid \leftarrow$PickOne(AllMax($\boldsymbol{W}$))
else
    $pid \leftarrow$FindReq($\boldsymbol{R}$)
end if
if $pid \neq$ NULL then
    $\boldsymbol{G}[pid] \leftarrow \top$
    $\boldsymbol{W}[pid] \leftarrow 0$
end if
return $\boldsymbol{G}$
```

```
Global: $\boldsymbol{I}[1..n]$ an array of idle-times
Procedure LongestIdle
Input: $\boldsymbol{R}[1..n]$ an array of requests
Output: $\boldsymbol{G}[1..n]$ an array of responses
$\boldsymbol{G} \leftarrow (\bot, \ldots, \bot)$
for $i \leftarrow 1$ to $n$ do
    if $\boldsymbol{R}[i] = \bot$ then
        $\boldsymbol{I}[i] \leftarrow \boldsymbol{I}[i] + 1$
    end if
end for
if IsRace($\boldsymbol{R}$) then
    $pid \leftarrow$PickOne(AllMax($\boldsymbol{I}$))
else
    $pid \leftarrow$FindReq($\boldsymbol{R}$)
end if
if $pid \neq$ NULL then
    $\boldsymbol{G}[pid] \leftarrow \top$
    $\boldsymbol{I}[pid] \leftarrow 0$
end if
return $\boldsymbol{G}$
```

Fig. 5: First Come First Serve and Longest Idle Priority Arbiters.

3 Information Leakage in Arbiter Protocols

We consider a system, which accepts a public input (also referred as the low security input) L, a secret input (or the so-called private, high-security input) H, and produces an observable output O. The model includes an *adversary*, the *malicious user* $\mathcal{A}$. The adversary invokes the system with the input L and observes the output O. $\mathcal{A}$ does not have direct access to the secret H, but would like to learn its value. Before invoking the system, $\mathcal{A}$ has some initial uncertainty about the value of H, while after observing O, some amount of information is leaked, thereby reducing $\mathcal{A}$'s uncertainty about H.

In our model, we consider three types of processes (1) an adversary controlled process, denoted by $P_{\mathcal{A}}$, (2) a process belonging to the victim, denoted by $P_{\mathcal{V}}$ ($P_{\mathcal{V}} \neq P_{\mathcal{A}}$), and (3) a benign process introduced as additional unpredictable behavior to the system. The adversary can observe only permission responses issued by the arbiter on his/her requests, denoted by $\boldsymbol{R}_{\mathcal{A}} = \{R_{\mathcal{A}1}, R_{\mathcal{A}2}, \ldots, R_{\mathcal{A}k}\}$, with the aim to gain as much information as possible on the permission requests of the victim's process $\boldsymbol{R}_{\mathcal{V}} = \{R_{\mathcal{V}1}, R_{\mathcal{V}2}, \ldots, R_{\mathcal{V}k}\}$. We consider the secret $H = \boldsymbol{R}_{\mathcal{V}}$ to be the list of permission requests of the victim process. The low security input to the system $L = \boldsymbol{R}_{\mathcal{A}}$ is the adversary-controlled data—the permission requests placed by the adversary process. The corresponding permission grants received by the adversary on his/her own requests, denoted by $\boldsymbol{G}_{\mathcal{A}} = \{G_{\mathcal{A}1}, G_{\mathcal{A}2}, \ldots, G_{\mathcal{A}k}\}$, are the data observed by the adversary $O = \boldsymbol{G}_{\mathcal{A}}$ (referred as the *observations*).

In this work, we quantify and compare the amount of maximal expected leakage the adversary can obtain for arbiter protocols presented in Section 2 considering possible choices of $P_{\mathcal{A}}$ and $P_{\mathcal{V}}$. This is a QIF analysis problem through the *main channel*, when the adversary observes the direct output of the system (i.e. his/her own access grant pattern). If the adversary can also observe non-functional aspects of the system behavior (e.g. the time it takes to respond to a request, or the power consumed) through a *side channel*, then one would also take those observations into account to quantify the information leakage through such side-channels.

3.1 Quantifying Information Leakage Using Entropy

Intuitively, the amount of information gained by the adversary is the difference between the initial uncertainty about the secret and the remaining uncertainty [25]. The field of QIF formalizes this intuitive statement by casting the problem in the language of *information theory*. Information theory uses the concept of *entropy* for the purpose of measuring the amount of information that can be transmitted over a channel, measuring information transmission in *bits of entropy*. Then, information entropy is used as a measurement of *uncertainty*.

We briefly define relevant information entropy measures here. Given a random variable X with a finite domain $\mathcal{X}$, and a variable Z that indexes the probabilities of X to take values $x \in \mathcal{X}$, denoted as $P(X = x \mid Z = z)$, the *information entropy* of X, denoted as $\mathcal{H}(X \mid Z = z)$, is given by

$$\mathcal{H}(X \mid Z = z) = \sum_{x \in \mathcal{X}} P(X = x \mid Z = z) \log_2 \frac{1}{P(X = x \mid Z = z)} \tag{1}$$

Let $\mathcal{Z}$ be the domain of Z. Given another random variable Y, over the domain $\mathcal{Y}$, and a conditional probabilities $P(X = x \mid Y = y, Z = z)$, also indexed by Z, the *conditional Shannon entropy of X given knowledge of Y* indexed by Z is

$$\mathcal{H}(X \mid Y, Z = z) = \sum_{y \in \mathcal{Y}} P(Y = y|Z = z) \mathcal{H}(X \mid Y = y, Z = z), \text{ where} \tag{2}$$

$$\mathcal{H}(X \mid Y = y, Z = z) = \sum_{x \in \mathcal{X}} P(X = x \mid Y = y, Z = z) \log_2 \frac{1}{P(X = x \mid Y = y, Z = z)} \tag{3}$$

We are interested in the maximal amount of information about X that could be learned given the knowledge of Y, as this describes the worst case leakage scenario. For this, we use *conditional Shannon entropy* and compute the maximal amount of the expected information gain as the difference of the initial uncertainty about X and the uncertainty after acquiring the knowledge of Y

$$\mathcal{I}(X, Y, Z) = \max_{z \in \mathcal{Z}} (\mathcal{H}(X \mid Z = z) - \mathcal{H}(X \mid Y, Z = z)) \tag{4}$$

In the context of QIF, we consider the public input L to be the *index* variable indexing probability distributions of the secret input H and the output O, with H and O being *random variables*. Thus, the above notations correspond to $Z = L$, $X = H$ and $Y = O$. A value of the input L along with the corresponding observation of the output O defines an *event* in the analysis.

To compute the expected maximal amount of information leaked, we need:

(i) **Initial uncertainty** the adversary has about the secret, $\mathcal{H}_{init}(H \mid L = l)$, for each of his/her inputs before making observations. This is computed following the Formula (1) using the initial probability distribution of the secret $P(H = h \mid L = l)$ conditioned by the adversary's inputs;

(ii) **Expected remaining uncertainty** about the secret, $\mathcal{H}_{fin}(H \mid O, L = l)$, over all observations the adversary can make after he/she provides an input l, computed as in (2): $\sum_{\omega \in \Omega} P(O = \omega \mid L = l) \mathcal{H}(H \mid O = \omega, L = l)$, where Ω is the domain of O, $P(O = \omega \mid L = l)$ is the probability of the adversary observing ω given the input l, and $\mathcal{H}(H \mid O = \omega, L = l)$ is the uncertainty about the secret given the event (ω, l), the latter computed using (3) and the probabilities of the secret conditioned by this event $P(H = h \mid O = \omega, L = l)$;

(iii) Then $\mathcal{I}(H, O, L) = \max(\mathcal{H}_{init}(H \mid L = l) - \mathcal{H}_{fin}(H \mid O, L = l))$ is the **expected maximal amount of information leaked**, as defined in (4).

These definitions formalize our intuition that the information leaked is the maximal difference between the uncertainty about the secret before and after making an observation. The value of the adversary's input L for which the maximal leakage is obtained defines the best strategy for the adversary to follow in order to obtain the maximal information leakage on H.

3.2 Extracting Observation Constraints with Symbolic Execution

Symbolic execution is a technique that extracts path constraints from a system by executing it on *symbolic* inputs, as opposed to concrete input values. It can be used to extract a set of path constraints characterizing all possible execution paths of the system (typically up to an execution depth bound).

We adopt and extend symbolic execution techniques to automatically extract constraints that relate secret values with observations that an adversary can make. Traditional symbolic execution does not focus on extracting constraints on observations that can be made by an adversary, such as timing or power measurements, or constraints on resources that can be shared with adversarial processes. To formalize this concept, we introduce *event constraints* of the protocol as defined below.

Let $\phi(H, L)$ be a path constraint returned by a traditional symbolic execution tool. Consider the set of observations Ω for the observable O. In practice multiple execution paths may map to the same observation. We assume, however, that each execution path maps to a single observation. To express this, we define a function $\mathcal{O}$, where $\mathcal{O}(\phi(H, L))$ is the observation that the execution path constraint $\phi(H, L)$ maps to. Then, we extend each path constraint $\phi(H, L)$ into an *event constraint* $\mathbb{C}_\phi(H, O, L)$ to pair it with the observation it yields to:

$$\mathbb{C}_\phi(H, O, L) : (O = \omega) \wedge \phi(H, L), \quad \text{where} \quad \mathcal{O}(\phi(H, L)) = \omega \tag{5}$$

The disjunction of all event constraints with the the same observation ω, characterizes ω by a constraint $\mathbb{C}_\omega(H, O, L)$ that holds if and only if the observation ω occurs, and can be written in the form:

$$\mathbb{C}_\omega(H, O, L) = \bigvee_{\mathcal{O}(\phi(H,L))=\omega} \wedge (O = \omega)\phi(H, L) \tag{6}$$

We define a *characteristic constraint* $\mathbb{C}(H, O, L)$ for the protocol as the constraint that describes all possible events:

$$\mathbb{C}(H, O, L) : \bigvee_{\omega \in \Omega} \mathbb{C}_\omega(H, O, L). \tag{7}$$

Example. Let us use the PRIORITY arbiter as a running example. For a single round $\Omega = \{\top, \bot\}$. We give the characteristic constraint for a single round of a three-process PRIORITY arbiter where $P_{\mathcal{A}} = P_2$ below:

$$\mathbb{C} = \mathbb{C}_\top \ \vee \ \mathbb{C}_\bot :$$

$$((O = \top) \ \wedge \ (R_1 = \bot \ \wedge \ R_2 = \top)) \vee$$
$$\vee \ ((O = \bot) \ \wedge \ (R_1 = \top \ \vee \ (R_1 = \bot \ \wedge \ R_2 = \bot \ \wedge \ R_3 = \top) \ \vee \ (R_1 = \bot \ \wedge \ R_2 = \bot \ \wedge \ R_3 = \bot))$$

3.3 Extension for the Symbolic Analysis of Random Components

Handling random components in symbolic analysis is a challenging task on its own. The first work on supporting random instances in symbolic execution has been introduced recently [18]. We propose a technique simulating randomness of symbolic variables that is well-fitted for quantitative analysis and is simpler. Since our approach is based on computing probabilities of protocol behaviors (i.e. the probabilities of the protocol following corresponding execution paths), we should take into account the distribution of random variables occurring in this protocol, and thus, in the path constraints. If a path constraint contains a random variable R, the probability of triggering that path depends on the probability of R taking specific values defined by the path.

To incorporate the probability distribution of R into the computation of the probabilities of the execution paths, we introduce a fresh symbolic integer variable *sym_R* and implement the PICKRAND() procedure in a way that it simulates the desired random generator behavior and extends path constraints to reflect the relation between *sym_R* and R as follows: each value r of R leads to multiple values of *sym_R* representing the weight of r in the probability distribution of R. Let R take values in $(R_1, \ldots, R_n)$ with probability weights

$\boldsymbol{W} = (W_1, \ldots, W_n)$, each $W_i \in \mathbb{Z}^+$. PickRnd() takes $\boldsymbol{W}$ for input and returns a value of R selected in a weighted-random manner in accord with $\boldsymbol{W}$. For each W_i, we define a *domain interval* $\mathcal{D}(W_i)$ of the length W_i as

$$\mathcal{D}(W_i) = \begin{cases} [1, W_i], & i = 1 \\ \big(\sum_{j=1}^{i-1} W_j, \sum_{j=1}^{i} W_j\big], & 1 < i \le n \end{cases} \tag{8}$$

We restrict *sym_R* to take values in non-empty domain intervals by instrumenting the code with the implementation of PickRnd() as given in Figure 6. If all domain intervals are empty, we set *sym_R* = NULL.

```
Global: $sym\_R$ a symbolic integer variable
Procedure PickRnd
Input: $\boldsymbol{W}[1..n]$ an array of weights
Output: $id$ an ID of a randomly selected value $R_{id}$
1: for $id \leftarrow 1$ to $id \le n$ do
2:     if $\boldsymbol{W}[id] > 0$ and $sym\_R \in \mathcal{D}(\boldsymbol{W}[id])$ then
3:         return $id$
4:     end if
5: end for
6: return NULL
```

Fig. 6: Selecting a value from a domain with a weighted-random distribution.

3.4 Computing Event Probabilities with Model Counting

In order to compute information leakage, we need to compute the probabilities given in (2) and (3). We compute the probability of an event by counting the number of values that satisfy the observation constraint (i.e., the number of solutions to the observation constraint) that corresponds to that event. To formalize this, we will use the following notations. Given an ordered set of variables $\boldsymbol{V}$ and an ordered subset $\boldsymbol{V'} \subseteq \boldsymbol{V}$,we define a partial assignment on $\boldsymbol{V'}$ as a mapping $\boldsymbol{V'} \mapsto \boldsymbol{v}$, where $\boldsymbol{v}$ is an assignment on all variables in $\boldsymbol{V'}$. Given a constraint $\Psi(\boldsymbol{V})$, we denote by $\Psi(\boldsymbol{V}) \mid_{\boldsymbol{V'} \mapsto \boldsymbol{v}}$ the result of assigning and propagating the values $\boldsymbol{v}$ to the variables $\boldsymbol{V'}$ in Ψ. We denote by $\#\Psi(\boldsymbol{V}) \mid_{\boldsymbol{V'} \mapsto \boldsymbol{v}}$ the number of solutions to $\Psi(\boldsymbol{V}) \mid_{\boldsymbol{V'} \mapsto \boldsymbol{v}}$ over the free variables.

Then the probabilities in (2) and (3) are computed using model counting on observation constraints as follows:

$$P(O = \omega | L = l) = \frac{\#\mathbb{C}(H, O, L) \mid_{(O,L) \mapsto (\omega, l)}}{\#\mathbb{C}(H, O, L) \mid_{(L) \mapsto (l)}} \tag{9}$$

$$P(H = h \mid O = \omega, L = l) = \frac{\#\mathbb{C}(H, O, L) \mid_{(H,O,L) \mapsto (h,\omega, l)}}{\#\mathbb{C}(H, O, L) \mid_{(O,L) \mapsto (\omega, l)}} \tag{10}$$

Example. In Table 1, we give the probability and entropy computations for the Priority arbiter when $P_{\mathcal{A}} = P_2$ and $P_{\mathcal{V}} = P_1$. We follow the computation steps described in Section 3.1 using (1)-(4) for the entropy calculations and (9), (10) for the probabilities.

$\boldsymbol{r_2}$	0				1			
$\boldsymbol{g_2}$	0		1		0		1	
$\boldsymbol{r_1}$	0	1	0	1	0	1	0	1
$\#\mathbb{C}\mid_{(R_1,G_2,R_2)\mapsto(r_1,g_2,r_2)}$	2	2	0	0	0	2	2	0
$\#\mathbb{C}\mid_{(G_2,R_2)\mapsto(g_2,r_2)}$	4		0		2		2	
$\#\mathbb{C}\mid_{(R_2)\mapsto(r_2)}$	4				4			
$\boldsymbol{P(R_1 = r_1 \mid R_2 = r_2)}$	$P(0 \mid 0) = 1/2$ $P(1 \mid 0) = 1/2$				$P(0 \mid 1) = 1/2$ $P(1 \mid 1) = 1/2$			
$\boldsymbol{\mathcal{H}_{init}(R_1 \mid R_2 = r_2)}$	1				1			
$\boldsymbol{P(R_1 = r_1 \mid G_2 = g_2, R_2 = r_2)}$	1/2	1/2	0	0	0	1	1	0
$\boldsymbol{\mathcal{H}(R_1 \mid G_2 = g_2, R_2 = r_2)}$	1		0		0		0	
$\boldsymbol{P(G_2 = g_2 \mid R_2 = r_2)}$	1		0		1/2		1/2	
$\boldsymbol{\mathcal{H}_{fin}(R_1 \mid G_2, R_2 = r_2)}$	1				0			
$\boldsymbol{\Delta\mathcal{H}}$	0				1			
$\boldsymbol{max(\Delta\mathcal{H})}$	1							

Table 1: Probability and entropy computations for the PRIORITY arbiter. Adversary controls P_2. Victim's process is P_1. $\mathbb{C}$ is the characteristic constraint for PRIORITY.

4 Model Counter for Arbiter Protocol Constraints

We observed that constraints extracted with the symbolic execution of the arbiter protocols were, on one hand, large—especially for those with random components as the randomization increases the variety of behaviours of the protocols, with over five million distinguished protocol behaviours for 6 rounds.

On the other hand, we observed that the constraints extracted from arbiter protocols can be characterized by a common structure. We define a grammar representing this structure, as described in Figure 7, and refer to its language as a *range constraint language*, denoted by $\mathcal{L}_{\mathcal{RC}}$. In the context of the constraints extracted with the extended symbolic execution: $\mathcal{B}$ stands for the Boolean variables representing each process's requests in each round and corresponding arbiter responses, and $\mathcal{I}$ for the integer variables, one per round, responsible for random components of the protocols. For deterministic protocols the domain of $\mathcal{I}$ is empty. An atomic constraint C in this grammar represents a single event constraint $\mathbb{C}_\phi(H, O, L)$ (defined in (5)) extracted with the extended symbolic execution. Variables representing arbiter responses are always present in an atomic constraint. Consequently, the atomic constraints have disjoint sets of solutions.

$$\begin{array}{lcl} \mathcal{C} & \rightarrow & \mathcal{C} \wedge \mathcal{C} \mid \mathcal{R} \\ \mathcal{R} & \rightarrow & \mathcal{B} = \top \mid \mathcal{B} = \bot \mid \mathcal{I} \in [a, b] \end{array}$$

Fig. 7: Range Constraint Grammar. $\mathcal{B}$ ranges over Boolean, $\mathcal{I}$ over integer variables.

We need to compute $\#\mathbb{C}(H, O, L)\ \mid_{(H,O,L)\mapsto(h,\omega,l)}$ for each tuple (h, ω, l). Based on the above observation on event constraints in $\mathcal{L}_{\mathcal{RC}}$, we built an efficient exact model counter which is linear in time in the size of the input constraint. The model counting is performed during parsing of the constraint and uses only as much space as required to store the final counts. We give a pseudocode for our model counter in Figure 8, where TUPLES$(\mathbb{C}_\phi, P_\mathcal{A}, P_\mathcal{V})$ returns a set of all tuples $(\boldsymbol{h}, \boldsymbol{\omega}, \boldsymbol{l})$ of the partial assignments $(\boldsymbol{R}_\mathcal{V}, \boldsymbol{G}_\mathcal{A}, \boldsymbol{R}_\mathcal{A}) \mapsto (\boldsymbol{h}, \boldsymbol{\omega}, \boldsymbol{l})$ of $\mathbb{C}_\phi$.

Given $P_\mathcal{V}$ and $P_\mathcal{A}$, each $\mathbb{C}_\phi$ determines values $(\boldsymbol{h}, \boldsymbol{\omega}, \boldsymbol{l})$ for (H, O, L), thus contributes to model counting for the tuple $(\boldsymbol{h}, \boldsymbol{\omega}, \boldsymbol{l})$. We define a *free variable* in an atomic constraint $\mathbb{C}_\phi$ to be a Boolean variable from the domain of $\mathcal{B}$ as a variable (i) distinguished from $\boldsymbol{R}_\mathcal{V}$ and $\boldsymbol{R}_\mathcal{A}$; and (ii) not appearing in $\mathbb{C}_\phi$. An

```
Global: $\boldsymbol{S}$ a data structure for storing model counts
Procedure $\mathcal{L}_{\mathcal{RC}}$_ModelCounter
Input: $\mathbb{C}$ a characteristic constraint, $P_{\mathcal{V}}$ victim's process, $P_{\mathcal{A}}$ adversary's process
Output: Model counts stored in $\boldsymbol{S}$
 1: for each $\mathbb{C}_\phi$ in $\mathbb{C}$ do                ▷ Also, by construction $\mathbb{C}_\phi \in \mathcal{L}_{\mathcal{RC}}$
 2:     $m \leftarrow$ #FreeVars($\mathbb{C}_\phi, P_{\mathcal{V}}, P_{\mathcal{A}}$)
 3:     $s \leftarrow 2^m$
 4:     for each $(r \in [a,b])$ in $I$ do
 5:         $s \leftarrow (b - a + 1) \times s$
 6:     end for
 7:     for each tuple $(\boldsymbol{h}, \boldsymbol{\omega}, \boldsymbol{l})$ in Tuples($\mathbb{C}_\phi, P_{\mathcal{V}}, P_{\mathcal{A}}$) do
 8:         $\boldsymbol{S}[(\boldsymbol{h}, \boldsymbol{\omega}, \boldsymbol{l})] \leftarrow \boldsymbol{S}[(\boldsymbol{h}, \boldsymbol{\omega}, \boldsymbol{l})] + s$
 9:     end for
10: end for
```

Fig. 8: Model counter for range constraints.

event constraint $\mathbb{C}_\phi$ in $\mathbb{C}$ contributes towards the model-counting of multiple tuples (equally, with the same number of models s) when any of the variables $\boldsymbol{R}_{\mathcal{V}}$ and $\boldsymbol{R}_{\mathcal{A}}$ is absent in $\mathbb{C}_\phi$. The number of models, s, depends only on the number of free variables and the ranges on the integer variables in $\mathbb{C}_\phi$.

5 Experiments

To test our framework, we conduct quantification experiments on nine different arbiter protocols discussed in Section 2, considering both stateless and stateful processes. Each experiment involves a single arbiter protocol, three processes, and rounds from one to six. We compute the maximum expected information leakage the adversary can learn about the victim process, and determine the position of the victim-adversary processes for which the arbiter leaks the most.

Our current implementation requires specification of each arbiter protocol in Java. We use SPF (Symbolic Java Pathfinder) [23], a well-established symbolic execution tool to analyze Java bytecode, to extract characteristic constraints for the arbiter protocols, as discussed in Section 3.2. Then, we perform model counting as explained in Sections 3.4 and 4. Based on the distribution of these counts, we calculate the information leakage according to Section 3.1.

We perform model counting with two methods: an enumerative counting method $\mathcal{EC}$ (Section 3.4), and our faster range-constraint counting method $\mathcal{RC}$ (Section 4). The former provides us a slow method serving as a ground truth, the latter an optimized method for higher numbers of rounds when the exponential blowup makes enumerative counting infeasible. Table 2 shows the execution time, in seconds, for $\mathcal{EC}$ vs $\mathcal{RC}$ methods. $\mathcal{RC}$ ranges from 1.4x faster to $2,647$x faster, with an average speedup of 250x (excluding time outs for $\mathcal{EC}$).

Figure 9 shows the results of our experiments, executed on a 128 GB RAM machine. The protocols are given in two groups: one with stateless processes, one of stateful processes. The leakage for each protocol is shown for each arrangement of $(victim, adversary)$ process IDs and six rounds of data; six horizontal lines in each bar delineate the information learned up through that round. The full bar is the information learned in six rounds; the lowest line is the information

Protocol	**1 Round**			**2 Rounds**			**3 Rounds**			**4 Rounds**			**5 Rounds**			**6 Rounds**		
	max *bit*	$\mathcal{RC}$ *sec*	$\mathcal{EC}$ *sec*	*max* *bit*	$\mathcal{RC}$ *sec*	$\mathcal{EC}$ *sec*	*max* *bit*	$\mathcal{RC}$ *sec*	$\mathcal{EC}$ *sec*	*max* *bit*	$\mathcal{RC}$ *sec*	$\mathcal{EC}$ *sec*	*max* *bit*	$\mathcal{RC}$ *sec*	$\mathcal{EC}$ *sec*	*max* *bit*	$\mathcal{RC}$ *sec*	$\mathcal{EC}$ *sec*
Priority	1.00	0.1	0.3	2.00	0.2	0.7	3.00	0.2	10.2	4.00	0.3	346.4	5.00	0.5	-	6.00	1.5	-
RoundRobin	0.00	0.2	0.4	0.00	0.1	0.3	0.00	0.2	1.2	0.00	0.3	10.3	0.00	0.3	225.0	0.00	0.8	-
RoundRobinSkip	1.00	0.2	0.3	1.16	0.2	0.6	1.57	0.1	10.3	1.97	0.3	337.9	2.32	0.5	-	2.71	1.5	-
FCFS	1.00	0.2	0.3	1.27	0.2	1.2	1.86	0.4	53.2	2.16	0.7	-	2.71	4.8	-	3.02	44.1	-
LongestIdle	1.00	0.1	0.3	1.55	0.2	1.0	2.10	0.3	53.7	2.66	0.8	-	3.22	5.1	-	3.78	45.7	-
FCFS_R	0.13	0.1	3.2	0.27	0.3	11.5	0.45	0.5	439.1	0.64	4.9	-	0.83	74.3	-	1.02	1121.1	-
LongestIdle_R	0.05	0.1	2.7	0.21	0.1	10.0	0.40	0.4	241.8	0.58	1.9	-	0.76	19.5	-	0.92	200.3	-
Lottery	0.05	0.2	2.7	0.09	0.2	13.2	0.13	0.5	399.7	0.17	4.2	-	0.21	65.2	-	0.25	981.2	-
Random	0.05	0.1	4.8	0.10	0.2	10.6	0.15	0.5	372.2	0.20	4.2	-	0.24	66.2	-	0.29	983.1	-
Priority_S	1.00	0.1	0.3	2.00	0.2	0.9	3.00	0.3	18.9	4.00	0.4	-	5.00	0.8	-	6.00	4.4	-
RoundRobin_S	0.00	0.1	0.3	0.00	0.2	0.5	0.00	0.3	5.2	0.00	0.3	260.8	0.00	0.4	-	0.00	1.2	-
RoundRobinSkip_S	1.00	0.2	0.4	1.07	0.1	1.1	1.33	0.2	17.6	1.53	0.4	979.5	1.64	0.8	-	1.81	3.2	-
FCFS_S	1.00	0.1	0.4	1.16	0.1	1.0	1.41	0.3	32.4	1.67	0.4	-	1.83	1.2	-	2.06	6.4	-
LongestIdle_S	1.00	0.2	0.4	1.55	0.2	1.2	2.14	0.3	36.7	2.78	0.4	-	3.47	1.3	-	4.20	6.6	-
FCFS_RS	0.13	0.2	4.3	0.25	0.1	17.3	0.41	0.4	283.2	0.55	1.3	-	0.70	9.5	-	0.84	79.2	-
LongestIdle_RS	0.05	0.2	4.1	0.14	0.2	15.7	0.31	0.3	184.0	0.35	0.6	-	0.43	3.1	-	0.48	20.5	-
Lottery_S	0.05	0.2	4.6	0.06	0.3	22.1	0.06	0.4	312.6	0.07	1.2	-	0.08	10.2	-	0.09	88.2	-
Random_S	0.05	0.1	2.9	0.06	0.2	18.8	0.08	0.3	290.8	0.09	1.3	-	0.10	10.2	-	0.11	88.9	-

Table 2: Max leakage (in bits) and execution time (in seconds) for leakage computation with the Range-constraint Counting ($\mathcal{RC}$) vs Enumerative Counting ($\mathcal{EC}$) methods. A timeout of 20 minutes (1200 s) was used. '-' indicates a timeout; (S) – stateful processes; (R) – resolving wait-time and idle-time concurrences randomly.

learned in the first round. The worst-case leakage of each protocol across all process pairs, for each round, is shown in Figure 10, which illustrates interesting trends and groupings among the protocols.

The variety of interesting subtleties in the results are more than we can discuss here, but we note a few points. The arrangement $(1, 2)$ is the best scenario for the attacker, as he/she directly follows the victim and no other processes cause noise. The Priority arbiter leaks the most for this arrangement, but leaks less for $(2, 3)$ and $(1, 3)$, and does not leak for other arrangements. The RoundRobin protocol leaks no information in any arrangement, but it is inefficient with respect to resource usage since it wastes cycles where the resource

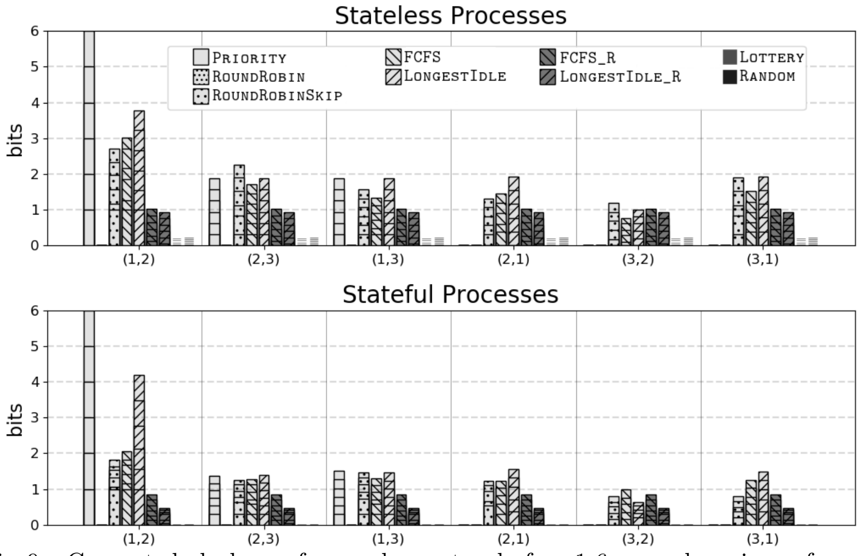

Fig. 9: Computed leakage for each protocol for 1-6 rounds, given for each $(victim, adversary)$ process pair. Cumulative leakage is shown for 6 rounds.

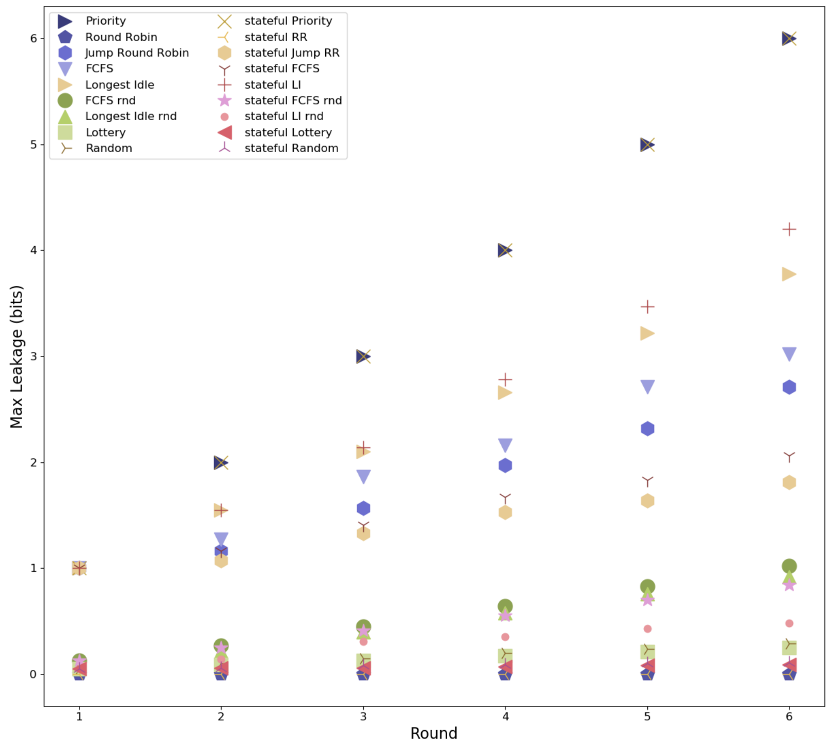

Fig. 10: Worst-case leakage of each protocol as a function of the round number.

is not utilized. Introducing a simple optimization in the RoundRobinSkip protocol improves resource usage, but introduces leakage. The random protocols (Lottery and Random) have low leakage, but they are non-deterministic protocols in how they award resources which can lead to unfair resource allocation. Introducing randomness to other algorithms, like FCFS and LongestIdle improve their leakage characteristics (again, at the expense of non-determinism). Typically, the stateful process version of each protocol leaks slightly less than the stateless version, as processes have less freedom in choosing their requests which means that there is less amount of information (entropy) to leak.

6 Related Work

Arbiter protocols have been studied intensively for effectiveness and fairness ([11] gives a brief survey). Various arbitration techniques have been proposed and compared in providing fairness and efficiency for shared-resource access management. More recent work has been focusing on privacy aspects of the arbitration, covert channel and timing side channel information leakage, including quantitative leakage analysis and channel capacity evaluations [4, 14, 24, 9]. However, these approaches are either manual, or consider a fixed number of processes and rounds, or focus on deterministic arbiters.

We make use of concepts from foundational and theoretical works in quantitative information flow [25] and combine them with symbolic execution and model counting techniques to automatically quantify security vulnerabilities in protocols. There are other model counting techniques that handle constraints with different levels of expressiveness [1, 17], and they can be integrated with the quantitative information flow analysis we present in this paper. Quantitative measurement of information leakage in programs has been an active area of research [6, 2, 26, 15]. Most previous works quantify the leakage in a single run of the program given a concrete value of low input. There have been recent works for performing automatic QIF for programs using symbolic execution [21, 20], bounded model checking [12], and graph theoretic methods [19], or random sampling [5], as well as in detecting and quantifying information flow and timing side channels at the hardware design and specification level [10, 8]. Multi-run analyses based on input enumeration [16] and symbolic approaches [22] have also been proposed for side-channel attack synthesis.

7 Conclusion

Contention for shared resources will only grow with time as we become increasingly reliant on multi-tenant, cloud systems. Isolation and privacy preservation are of the utmost importance in these systems, but virtual machines and OS guards cannot always prevent information from crossing from one domain to another. Adversaries can use information leakages to extrapolate privileged information that needs to remain secure. The novel QIF analysis technique in this paper combines and extends symbolic execution and model counting techniques providing protocol designers and users a new dimension in assessment and comparison of protocols in terms of the amount of information leaked over time.

References

1. Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. Automata-based model counting for string constraints. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24*, 2015.
2. Michael Backes, Boris Kopf, and Andrey Rybalchenko. Automatic Discovery and Quantification of Information Leaks. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, SP '09, Washington, DC, USA, 2009.
3. Michael Backes and Birgit Pfitzmann. Computational probabilistic noninterference. *Int. J. Inf. Sec.*, 3(1):42–60, 2004.
4. Serdar Cabuk, Carla E. Brodley, and Clay Shields. IP covert channel detection. *ACM Trans. Inf. Syst. Secur.*, 12(4):22:1–22:29, 2009.
5. Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. Leakwatch: Estimating information leakage from java programs. In *Computer Security - ESORICS'14 - 19th European Symposium on Research in Computer Security. Proceedings*, 2014.
6. David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *J. Comput. Secur.*, 15(3):321–371, August 2007.
7. Roya Ensafi, Jong Chun Park, Deepak Kapur, and Jedidiah R. Crandall. Idle port scanning and non-interference analysis of network protocol stacks using model checking. In *19th USENIX Security Symposium, Washington, DC, USA*, 2010.
8. Andrew Ferraiuolo, Weizhe Hua, Andrew C. Myers, and G. Edward Suh. Secure information flow verification with mutable dependent types. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC 2017.

9. Xun Gong and Negar Kiyavash. Quantifying the information leakage in timing side channels in deterministic work-conserving schedulers. *IEEE/ACM Trans. Netw.*, 24(3):1841–1852, 2016.
10. Shengjian Guo, Meng Wu, and Chao Wang. Symbolic execution of programmable logic controller code. In *ESEC/SIGSOFT FSE*, 2017.
11. J. Gupta and N. Goel. Efficient bus arbitration protocol for soc design. In *2015 International Conference on Smart Technologies and Management for Computing, Communication, Controls, Energy and Materials (ICSTM)*, 2015.
12. Jonathan Heusser and Pasquale Malacaria. Quantifying information leaks in software. In *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010*, pages 261–269, 2010.
13. Dominic J. D. Hughes and Vitaly Shmatikov. Information hiding, anonymity and privacy: a modular approach. *Journal of Computer Security*, 12(1):3–36, 2004.
14. Sachin Kadloor and Negar Kiyavash. Delay optimal policies offer very little privacy. In *Proceedings of the IEEE INFOCOM 2013, Turin, Italy, 2013.*
15. Vladimir Klebanov, Norbert Manthey, and Christian Muise. SAT-Based Analysis and Quantification of Information Flow in Programs. In *Quantitative Evaluation of Systems*, volume 8054 of *LNCS*, pages 177–192. Springer Berlin Heidelberg, 2013.
16. Boris Köpf and David A. Basin. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA*, 2007.
17. Jesús A. De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. Effective lattice point counting in rational convex polytopes. *J. Symb. Comput.*, 38(4):1273–1302, 2004.
18. Pasquale Malacaria, M. H. R. Khouzani, Corina S. Pasareanu, Quoc-Sang Phan, and Kasper Søe Luckow. Symbolic side-channel analysis for probabilistic programs. *IACR Cryptology ePrint Archive*, 2018:329, 2018.
19. Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA*, 2008.
20. Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. Synthesis of adaptive side-channel attacks. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA*, 2017.
21. Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S. Pasareanu. Symbolic quantitative information flow. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.
22. Corina S. Păsăreanu, Quoc-Sang Phan, and Pasquale Malacaria. Multi-run side-channel analysis using Symbolic Execution and Max-SMT. In *29th IEEE Computer Security Foundations Symposium*, CSF 2016, Washington, DC, USA, 2016.
23. Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *ASE*, 2013.
24. S. H. Sellke, Chih-Chun Wang, N. E. Shroff, and Sonchita Bagchi. Capacity bounds on timing channels with bounded service times. *2007 IEEE International Symposium on Information Theory*, pages 981–985, 2007.
25. Geoffrey Smith. On the foundations of quantitative information flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS)*, pages 288–302, 2009.
26. Chao Wang and Patrick Schaumont. Security by compilation: An automated approach to comprehensive side-channel resistance. *ACM SIGLOG News*, 4(2):76–89, May 2017.