2018 IEEE Symposium on Security and Privacy

Static Evaluation of Noninterference using Approximate Model Counting

Ziqiao Zhou
University of North Carolina
Chapel Hill, NC, USA
ziqiao@cs.unc.edu

Zhiyun Qian
University of California
Riverside, CA, USA
zhiyunq@cs.ucr.edu

Michael K. Reiter
University of North Carolina
Chapel Hill, NC, USA
reiter@cs.unc.edu

Yinqian Zhang
The Ohio State University
Columbia, OH, USA
yinqian@cse.ohio-state.edu

***Abstract*—Noninterference is a definition of security for secret values provided to a procedure, which informally is met when attacker-observable outputs are insensitive to the value of the secret inputs or, in other words, the secret inputs do not "interfere" with those outputs. This paper describes a static analysis method to measure interference in software. In this approach, interference is assessed using the extent to which different secret inputs are consistent with different attacker-controlled inputs and attacker-observable outputs, which can be measured using a technique called *model counting*. Leveraging this insight, we develop a flexible interference assessment technique for which the assessment accuracy quantifiably grows with the computational effort invested in the analysis. This paper demonstrates the effectiveness of this technique through application to several case studies, including leakage of: search-engine queries through auto-complete response sizes; secrets subjected to compression together with attacker-controlled inputs; and TCP sequence numbers from shared counters.**

***Index Terms*—information flow; noninterference; approximate model counting**

I. Introduction

Information leakage about secrets in software is a core concern for computer security, and has been for decades (e.g., [1]). Leakage can in principle be detected by tracking information flow from secret objects to attacker-observable ones, and considerable progress has been made on static-analysis tools for detecting leakage vulnerabilities in software (see Sec. II for a discussion of related work). Still, however, assessing the significance of detected leaks continues to be a difficulty that plagues static-analysis tools, particularly for ones that track implicit information flows (e.g., [2]).

In this paper we propose a static-analysis method to assess leakage vulnerabilities, even those that leverage implicit flows. The intuition of our design draws from *noninterference* [3], which informally is achieved when the attacker-controlled inputs and attacker-observable outputs are unchanged by the value of a secret input that should not "interfere" with what the attacker can observe. In practice, noninterference is extremely unlikely to hold for most real-world programs, since a degree of leakage is often necessary. As such, a more quantitative measurement of (non)interference should be more useful in assessing leakage. In principle, if all possible pairs of attacker-controlled inputs and attacker-observable outputs could be enumerated for any given value of the secret input, then differences in the pairs possible for different secrets would reveal interference between the secret value and the pairs that remain possible, and hence an estimate for potential leakage. Unfortunately, enumerating these pairs for all possible secret values is often impractical for complex procedures, and so previous explorations based on similar principles have been limited (again, see Sec. II).

Within this framework, we explore the assessment of leakage vulnerabilities by randomly sampling a space of secret values and then limiting our search for pairs of attacker-controlled inputs and attacker-observable outputs to only those that are consistent with some secret in that space. By leveraging techniques from *approximate model counting* [4], we show how to scalably estimate the number of such pairs to a desired accuracy and confidence and—perhaps more to the point—the number of such pairs that are consistent with one or both of two disjoint spaces of secret values. Finding two spaces of secret values for which these counts suggest pairs consistent with one but not both then reveals interference. Moreover, we will demonstrate the need to examine samples of secrets of varying sizes, and show how small samples can provide a more reliable indication of the *number* of secret values about which information leaks, while larger samples provide more insight into the *amount* of leakage of secret values. In doing so, we develop a powerful framework for interference detection and assessment with the following strengths:

- The error in our assessment of a reported interference can be reduced, arbitrarily close to zero in the limit, through greater computational investment. Specifically, by increasing the accuracy and confidence with which the number of pairs of attacker-controlled inputs and attacker-observable outputs consistent with sampled secrets are estimated, and by increasing the number and variety of samples tested, the interference assessment quantifiably improves.
- Our framework supports the derivation of values from its estimates that can separately provide insight into the *number* of secret values about which information leaks, and the *amount* of leakage about those secrets. Within the context of particular applications, one type of leakage might be more important than the other.
- Even for nondeterministic applications, our framework provides a robust assessment of noninterference, by accounting for the nondeterministic factors (e.g., procedure inputs other than the secrets or attacker-controlled values).

DOI 10.1109/SP.2018.00052

We detail our approach and its implementation in a tool. Our tool takes as input a procedure; a specification of which of its inputs are attacker-controlled, which are secrets, and which outputs are attacker-observable; and parameters to drive the secret-sampling strategy to reach a desired confidence in its leakage assessment.

We demonstrate our tool through its application in numerous scenarios. We first apply it to selected, artificially small examples (microbenchmarks) to demonstrate its features. Then, we apply it to assess leakage in several real-world examples.

- We apply our tool to detect leakage of web search query strings submitted to the `Sphinx` web server on the basis of auto-complete response sizes returned to the client (i.e., even if the query and response contents themselves are encrypted) [5]. We also leverage our tool to evaluate the impact of various mitigation strategies on this leak, e.g., showing that based on the contents of the searchable database, some seemingly stronger defenses offer little additional protection over seemingly weaker ones.
- We use our tool to demonstrate the vulnerability leveraged in CRIME attacks [6], specifically that adaptive compression algorithms provide opportunities for an attacker to test guesses about secrets that he cannot observe, if he can instead observe the length of compressed strings containing both the secret and his guess. This case study demonstrates the ability of our technique to effectively account for attacker-controlled inputs, in contrast to many prior techniques (see Sec. II). Specifically, we apply our tool to both `Gzip` and the fixed-dictionary compression library `Smaz` to illustrate that they both leak information about secrets to the adversary, but that `Gzip` leaks more information as the number of adversary-controlled executions grows.
- We apply our tool to illustrate the tendency of Linux to leak TCP-session sequence numbers to an off-path attacker [7], [8]. This is perhaps the most complex of the examples we consider, and again illustrates the power of accounting for attacker-controlled variables. Moreover, we evaluate two plausible defenses against this attack, one a hypothetical patch to Linux that we propose, and another being simply to disable use of information that is central to the leak.

This paper is structured as follows. We discuss related work in Sec. II and then present our methodology for interference measurement in Sec. III. The implementation of our tool is described in Sec. IV. We use microbenchmarks in Sec. V to demonstrate features of our approach, and then apply our tool to real-world codebases in Sec. VI. Some limitations of our approach are discussed in Sec. VII. We conclude in Sec. VIII.

II. Related Work

Our work can be viewed as a form of static information-flow analysis, an area with a long history of prior work (e.g., see [9]–[13] and the references therein). A central challenge (e.g., [2]) in this space is how to assess detected leaks, as some might be insufficiently consequential to warrant attention. One strategy that is often adopted is to simply ignore implicit flows (e.g., [14]). In contrast, our analysis encompasses both implicit and explicit flows.

A second approach to assess leaks, often termed *quantitative information flow* (QIF, e.g., [15]–[22]), is to compute the amount of information leaked about the secret (e.g., in terms of some type of entropy), conditioned on the output values observable by the attacker. In the context of static analysis, QIF has already been used to estimate leakage for cache side-channel attacks based on an abstract cache model (e.g., [23], [24]) and leakage from network traffic of web applications (e.g., [5], [25]–[27]), for example. To our knowledge, our work improves on prior work in QIF along one or more of the following dimensions. First, computing the measures in these works often involves computing outputs induced by possible secret values, which sometimes leverages application-specific restrictions to be tractable (e.g., [25]). Our framework, in contrast, does not require such application-specific restrictions. Second, exploiting leakage vulnerabilities often requires attackers not only to observe outputs but also to inject inputs, and many applications incorporate other inputs, as well. These QIF calculations are not possible without knowing the distributions from which these values are drawn (e.g., [28]), and so some works (e.g., [29], [30]) heuristically assign specific values to these unknown inputs, potentially hiding the leakage from other assignments. Our analysis computes conditionals in a different "direction," i.e., counting possible combinations of attacker-controlled inputs, other inputs, and attacker-observable outputs conditioned on sets of secret values. In doing so, our technique accommodates attacker-controlled inputs but does not presume knowledge of the attacker's strategy or the distributions of these or other inputs. Third, some QIF frameworks work only for deterministic procedures (e.g., [31], [32]), whereas ours accommodates nondeterministic ones, as well.

The tractability of our design derives in part from results in model counting (or #SAT) [33]. Previous QIF-related works leveraging model counting either support only convex constraints (e.g., [31], [34]) and so therefore do not capture all constraints of realistic applications, or use exact counts (e.g., [27]) and so cannot scale to complex applications. In contrast, we leverage principled sampling-based methods for approximate model counting, which we show can be used to expose leaks in real codebases. Our work also demonstrates a new approach for using model counting to estimate information leakage, again deriving from our strategy of counting pairs of attacker-controlled inputs and attacker-observable outputs conditioned on secret-value sets of different sizes.

More distantly related to our work are several that simply detect interference (or interference meeting certain constraints), versus measure it (e.g., [35]–[41]). With few exceptions (e.g., [41]), most such works are applied on an abstract model of a program, rather than working from off-the-shelf programs used in practice, as we do. Moreover, only *detecting* interference is arguably less useful when interference is necessitated by the program's goals but extraneous leakage should be otherwise measured and minimized.

III. Interference Assessment

Our technique seeks to measure information leakage from a procedure *proc*. The set $\mathit{Vars}_{\mathsf{O}}$ is the set of attacker-observable formal output parameters of *proc*; after completion, *proc* outputs a value $\mathsf{O}(\mathit{ovar})$ for each formal output parameter $\mathit{ovar} \in \mathit{Vars}_{\mathsf{O}}$. Outputs from *proc* that the attacker cannot observe are not modeled. The formal input parameters to *proc* are divided into three disjoint sets, namely $\mathit{Vars}_{\mathsf{C}}$, $\mathit{Vars}_{\mathsf{I}}$, and $\mathit{Vars}_{\mathsf{S}}$, having the following properties.

- Each formal parameter $\mathit{cvar} \in \mathit{Vars}_{\mathsf{C}}$ takes on a value $\mathsf{C}(\mathit{cvar})$ controlled by the attacker.
- Each formal parameter $\mathit{ivar} \in \mathit{Vars}_{\mathsf{I}}$ takes on a value $\mathsf{I}(\mathit{ivar})$ that is not controlled by the attacker.
- Each formal parameter $\mathit{svar} \in \mathit{Vars}_{\mathsf{S}}$ takes on a value $\mathsf{S}(\mathit{svar})$ that is not controlled by the attacker and moreover, represents a secret for which we are specifically concerned with detecting leakage via the outputs O.

So, for our purposes, we consider *proc* to be of the form

$$\mathsf{O} \leftarrow \mathit{proc}(\mathsf{C}, \mathsf{I}, \mathsf{S})$$

with O, C, I, and S assigning values to the formal parameters of *proc* as described above.

Execution of *proc* ensures a logical postcondition Π_{proc} that constrains how the variables represented in O, C, I, and S relate to one another. We denote this predicate instantiated with particular input and output values by $\Pi_{\mathit{proc}}(\mathsf{C}, \mathsf{O}, \mathsf{I}, \mathsf{S})$, which is either true or false.

To simplify discussion, we assume in this paper that there is only one secret formal parameter 'secret' (i.e., $\mathit{Vars}_{\mathsf{S}} = \{\text{'secret'}\}$), though our framework naturally extends to more. We assume that the value of 'secret' is chosen from a set $\mathbb{S}$, which the attacker knows. To measure the leakage about 'secret' from O, under the adversary's chosen C, we consider the set Y_s of pairs $\langle \mathsf{C}, \mathsf{O} \rangle$ that are consistent with $\mathsf{S}(\text{'secret'})$:

$$X_s = \left\{ \langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \;\middle|\; \Pi_{\mathit{proc}}(\mathsf{C}, \mathsf{O}, \mathsf{I}, \mathsf{S}) \wedge \mathsf{S}(\text{'secret'}) = s \right\}$$
$$Y_s = \{ \langle \mathsf{C}, \mathsf{O} \rangle \mid \exists \mathsf{I} : \langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \in X_s \}$$

In these definitions, the sets $\mathit{Vars}_{\mathsf{C}}$, $\mathit{Vars}_{\mathsf{I}}$, and $\mathit{Vars}_{\mathsf{O}}$ (and $\mathit{Vars}_{\mathsf{S}}$) are assumed to be fixed. For example, if $\langle \mathsf{C}, \mathsf{O} \rangle \in Y_s$ and $\langle \mathsf{C}', \mathsf{O}' \rangle \in Y_s$, then while C and C' (respectively, O and O') can differ in the values they assign to variables (e.g., $\mathsf{C}(\mathit{cvar}) \neq \mathsf{C}'(\mathit{cvar})$ for some cvar), they cannot differ on the variables to which they assign values.

The reason for considering Y_s is that it is an indicator of how s influences the possible view of the adversary, in terms of the variables it controls (C) and the variables it observes (O). For example, if O is independent of 'secret' and so leaks nothing about the value of 'secret', regardless of how the adversary chooses C, then $Y_s = Y_{s'}$ for any $s, s' \in \mathbb{S}$. To generalize from this example, let $Y_S = \bigcup_{s \in S} Y_s$ and then consider the Jaccard distance of Y_S and $Y_{S'}$ for any two disjoint sets $S, S' \subseteq \mathbb{S}$:

$$J(S, S') = \frac{\left|(Y_S \setminus Y_{S'}) \cup (Y_{S'} \setminus Y_S)\right|}{\left|Y_S \cup Y_{S'}\right|} = 1 - \frac{\left|Y_S \cap Y_{S'}\right|}{\left|Y_S \cup Y_{S'}\right|} \tag{1}$$

(By convention, $J(S, S') = 0$ if $Y_S = Y_{S'} = \emptyset$.) On the one hand, $J(S, S') = 0$ implies that $Y_S = Y_{S'}$ or, in other words, that an attacker cannot distinguish whether the secret $\mathsf{S}(\text{'secret'})$ is in S or S'. On the other hand, $J(S, S') > 0$ implies there is some $\langle \mathsf{C}, \mathsf{O} \rangle \in (Y_S \setminus Y_{S'}) \cup (Y_{S'} \setminus Y_S)$, and so the attacker can potentially distinguish between 'secret' having a value in S and the case in which it has a value in S'.

Unfortunately, it is generally infeasible to compute $J(S, S')$ for every disjoint pair $S, S' \subseteq \mathbb{S}$, or even when S, S' are restricted to being singleton sets. We can, however, estimate

$$J_n = \underset{\substack{S, S' : |S| = |S'| = n \\ \wedge\, S \cap S' = \emptyset}}{\operatorname{avg}} J(S, S') \tag{2}$$

to a high level of confidence by sampling disjoint sets S, S' of size n (or of expected size n, as we will discuss in Sec. IV-A) at random and computing $J(S, S')$ for each.

A. The need to vary n

Consider an idealized situation in which a procedure leaks the equivalence class into which $\mathsf{S}(\text{'secret'})$ falls, among a set of c "small" equivalence classes $\mathbb{C}_1, \ldots \mathbb{C}_c$ of equal size w. If $\mathbb{C} = \bigcup_{i=1}^{c} \mathbb{C}_i$, then the remaining elements $\mathbb{C}_0 = \mathbb{S} \setminus \mathbb{C}$ form another, "large" equivalence class ($w < |\mathbb{C}_0|$). Let $C_S^{\mathsf{sm}} \subseteq \{\mathbb{C}_1, \ldots, \mathbb{C}_c\}$ denote the small equivalence classes of which S contains elements and $C_S^{\mathsf{lg}} \subseteq \{\mathbb{C}_0\}$ indicate whether S contains representatives of $\mathbb{C}_0$ (in which case $C_S^{\mathsf{lg}} = \{\mathbb{C}_0\}$) or not (in which case $C_S^{\mathsf{lg}} = \{\}$). For simplicity, we assume below that $|Y_{\mathbb{C}_i}|$ is the same for each $i \in \{0, 1, \ldots, c\}$.

For the rest of this discussion, we treat the selection of $s \in S$ and $s \in S'$ as the selection, with replacement, of $\mathbb{C}_i \ni s$.[1] Then, $\mathbb{E}\left(\left|C_S^{\mathsf{sm}}\right|\right) = c\left(1 - \left(1 - \frac{w}{|\mathbb{S}|}\right)^n\right)$,[2] and so

$$\mathbb{E}\left(\left|C_S^{\mathsf{sm}} \cup C_{S'}^{\mathsf{sm}}\right|\right) = \mathbb{E}\left(\left|C_{S \cup S'}^{\mathsf{sm}}\right|\right) = c\left(1 - \left(1 - \frac{w}{|\mathbb{S}|}\right)^{2n}\right)$$
$$\mathbb{E}\left(\left|C_S^{\mathsf{sm}}\right| + \left|C_{S'}^{\mathsf{sm}}\right|\right) = 2c\left(1 - \left(1 - \frac{w}{|\mathbb{S}|}\right)^{n}\right)$$

Similarly,

$$\mathbb{E}\left(\left|C_S^{\mathsf{lg}} \cup C_{S'}^{\mathsf{lg}}\right|\right) = 1 - \left(\frac{cw}{|\mathbb{S}|}\right)^{2n}$$
$$\mathbb{E}\left(\left|C_S^{\mathsf{lg}}\right| + \left|C_{S'}^{\mathsf{lg}}\right|\right) = 2\left(1 - \left(\frac{cw}{|\mathbb{S}|}\right)^{n}\right)$$

[1] In reality, each $\mathbb{C}_i$ can be selected only w times in the drawing of S and S', since S and S' do not intersect. This dependence should not affect our estimates much, however, provided that w is not too small or n is small enough.

[2] Let $X_i = 1$ if class $\mathbb{C}_i \in C_S^{\mathsf{sm}}$ and $X_i = 0$ otherwise. Then, $\mathbb{P}(X_i = 0) = (1 - w/|\mathbb{S}|)^n$ and so $\mathbb{P}(X_i = 1) = 1 - (1 - w/|\mathbb{S}|)^n$. So, $\mathbb{E}\left(\left|C_S^{\mathsf{sm}}\right|\right) = \sum_{i=1}^{c} \mathbb{E}(X_i) = \sum_{i=1}^{c} \mathbb{P}(X_i = 1) = c\left(1 - \left(1 - \frac{w}{|\mathbb{S}|}\right)^n\right)$.

and so

$$\mathbb{E}\left(\left|C_S \cup C_{S'}\right|\right) = c\left(1 - \left(1 - \frac{w}{|\mathbb{S}|}\right)^{2n}\right) + 1 - \left(\frac{cw}{|\mathbb{S}|}\right)^{2n} \tag{3}$$

$$\mathbb{E}\left(\left|C_S\right| + \left|C_{S'}\right|\right) = 2\left(c\left(1 - \left(1 - \frac{w}{|\mathbb{S}|}\right)^{n}\right) + 1 - \left(\frac{cw}{|\mathbb{S}|}\right)^{n}\right) \tag{4}$$

Since $J_n = 1 - \frac{\left|C_S \cap C_{S'}\right|}{\left|C_S \cup C_{S'}\right|} = 2 - \frac{|C_S| + \left|C_{S'}\right|}{\left|C_S \cup C_{S'}\right|}$, we estimate $\mathbb{E}(J_n) \approx 2 - \frac{\mathbb{E}\left(|C_S| + \left|C_{S'}\right|\right)}{\mathbb{E}\left(\left|C_S \cup C_{S'}\right|\right)}$, using (4) and (3) for the numerator and denominator, respectively.

- First suppose n is small or, specifically, that $\frac{2nw}{|\mathbb{S}|} \ll 1$. Then, we can apply the binomial approximation $\left(1 - \frac{w}{|\mathbb{S}|}\right)^n \approx 1 - \frac{nw}{|\mathbb{S}|}$ to (4) and $\left(1 - \frac{w}{|\mathbb{S}|}\right)^{2n} \approx 1 - \frac{2nw}{|\mathbb{S}|}$ to (3) to conclude

$$\mathbb{E}(J_n) \approx 2 - \frac{\frac{2ncw}{|\mathbb{S}|} + 2 - 2\left(\frac{cw}{|\mathbb{S}|}\right)^n}{\frac{2ncw}{|\mathbb{S}|} + 1 - \left(\frac{cw}{|\mathbb{S}|}\right)^{2n}} \tag{5}$$

  Thus, when n is small, $\mathbb{E}(J_n)$ is sensitive to the number of secrets $cw = |\mathbb{C}|$ about which there is substantial leakage, but is insensitive to c and w individually, i.e., to the *amount* of leakage about those secrets. As such, small n yields a measure J_n that best indicates the *number* of secrets about which information leaks.
- Now suppose n is large, such that $\left(\frac{cw}{|\mathbb{S}|}\right)^n \approx 0$. Then,

$$\mathbb{E}(J_n) \approx 2 - \frac{2\left(c\left(1 - \left(1 - \frac{w}{|\mathbb{S}|}\right)^n\right) + 1\right)}{c\left(1 - \left(1 - \frac{w}{|\mathbb{S}|}\right)^{2n}\right) + 1} \tag{6}$$

  That is, J_n is sensitive to c and w individually when n is large. In this sense, we say that J_n for large n is a better indicator for the *amount* of leakage about secrets.

Again, the above model is idealized; leakage from real procedures can be far more complex. Still, this discussion provides insight into the utility of J_n and how it should be used. When n is small, (5) grows as $cw = |\mathbb{C}|$ grows, and for any threshold $t \in [0, 1]$ indicating "substantial" leakage, the smallest n for which $J_n \geq t$ shrinks. This smallest n is thus a reflection of $|\mathbb{C}|$, i.e., of the number of secrets about which information leaks. When n is large and for a fixed cw, (6) grows as w shrinks,[3] and for any threshold $t \in [0, 1]$ indicating "substantial" leakage, the largest n for which $J_n \geq t$ grows. This largest n is thus a reflection of w, i.e., of the amount of leakage about those secrets. It is therefore natural to examine both $\min\{n | J_n \geq t\}$ and $\max\{n | J_n \geq t\}$. To define measures using these values that fall within $[0, 1]$ and for which larger values indicate more leakage (as with J_n), we define

$$\eta_t^{\min} = \begin{cases} 0 & \text{if } t > J^{\max} \\ 1/\min\{n \mid J_n \geq t\} & \text{otherwise} \end{cases}$$

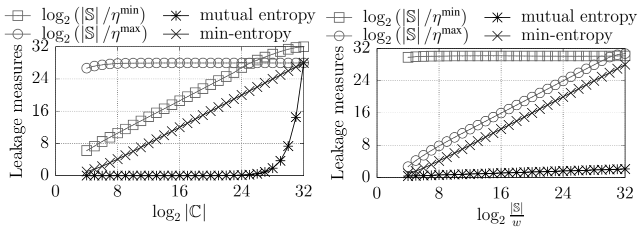
$$\eta_t^{\max} = \begin{cases} 0 & \text{if } t > J^{\max} \\ \frac{1}{|\mathbb{S}|/2}\max\{n \mid J_n \geq t\} & \text{otherwise} \end{cases}$$

Here, $J^{\max} = \max_{n'} J_{n'}$, and so the $t > J^{\max}$ cases accommodate t values larger than J_n ever reaches. Finally, rather than select a t to define "substantial" leakage, we simply take the average values of $\eta_t^{\min}$ and $\eta_t^{\max}$ over $t \in [0, 1]$ as our final measures:

$$\eta^{\min} = \int_0^1 \eta_t^{\min} dt \qquad \eta^{\max} = \int_0^1 \eta_t^{\max} dt \tag{7}$$

The numbers we report in this paper are discrete approximations to these values via numerical integration with a fixed subinterval width of 0.01.

Roughly speaking, a larger value for $\eta^{\min}$ suggests that information leaks from the procedure for more secret values, and a larger value for $\eta^{\max}$ suggests that more information leaks from the procedure about secret values.[4] To relate these measures to another used previously in the QIF literature, namely min-entropy (e.g., [42], [43]), in Fig. 1 we show $\eta^{\min}$ and $\eta^{\max}$ in comparison to the min-entropy of S('secret'), for our idealized setting above. Fig. 1(a) shows that $\eta^{\min}$ reflects the growth of $|\mathbb{C}|$ just as min-entropy can, and similarly, Fig. 1(b) shows that $\eta^{\max}$ reflects changes in w like min-entropy can. However, min-entropy does not distinguish between these types of leakage. Mutual entropy (e.g., [19], [22], [29]) also reflects increasing leakage as $|\mathbb{C}|$ grows in Fig. 1(a) and as w shrinks in Fig. 1(b), though its sensitivity to these effects is limited, particularly that of increasing $|\mathbb{C}|$, until $|\mathbb{C}|$ becomes quite large (Fig. 1(a)).



(a) Varying $|\mathbb{C}|$ with fixed $w = 2^4$ and $|\mathbb{S}| = 2^{32}$ (b) Varying w with fixed $|\mathbb{C}| = 2^{28}$ and $|\mathbb{S}| = 2^{32}$

Fig. 1: Relating $\eta^{\min}$ and $\eta^{\max}$ to min-entropy and mutual entropy, for the idealized model of leakage explored in Sec. III-A

[3]For example, $\left(1 - \frac{w}{|\mathbb{S}|}\right)^n < \frac{1}{2}$ is sufficient to ensure this.

[4]While these rules of thumb are accurate when J_n has no valley, they are less reliable when it does. In such cases, a more reliable understanding can be obtained by examining the graph of J_n directly, or at least by computing a separate $\eta^{\min}$ and $\eta^{\max}$ for each valley-free segment of J_n. Here, by "valley" we mean values n, n' where $n < n'$, $J_n > J_{n+1}$, $J_{n'} < J_{n'+1}$, and $J_{n''} = J_{n''+1}$ for each $n'' \in [n+1, n'-1]$. We have not encountered J_n curves with valleys in practice, and so do not discuss them further here.

B. Procedures with other inputs

The measures J_n, $\eta^{\min}$, and $\eta^{\max}$ are appropriate when *proc* is deterministic and leverages no inputs in I. When either of these restrictions are lifted, our approach described so far can be unreliable. We illustrate this in Sec. III-B1 and then provide an alternative measure in Sec. III-B2 that is more robust.

1) Limitations of J_n: First consider a randomized password checker that receives a secret password S('secret') and a candidate password C('test') and, for some constant $M > 0$, outputs a random value in $[0, M-1]$ if the candidate password is equal to the secret password and random value in $[M, M+16]$ otherwise. Intuitively, the leakage of this procedure should be the same as a deterministic password checker and independent of the value of M. However, as shown in Fig. 2, the use of randomness here results in an unintuitive result, since J_n (Fig. 2(b)) is sensitive to the value of M. As such, while our detector does accurately detect leakage in this case, it provides less help in comparing the leakage of two randomized implementations.

Another problem may arise when other inputs are allowed in I. Consider the example

```
proc (C, I, S)
    O('result') ← ((S('secret') > C('test')) ? 1 : 0)
                              ⊕ ((I('other') ≤ 0) ? 1 : 0)
    return O
```

Here, the expression "*cond* ? 1 : 0" evaluates to 1 if *cond* is true and 0 otherwise, and "$\oplus$" represents XOR. This procedure indicates that S('secret') $>$ C('test') by returning 0 if I('other') ≤ 0 or by returning 1 if I('other') > 0. Because our technique allows for any value of I('other') consistent with Π_{proc} when estimating $|Y_S|$, it will compute $J_n = 0$ for any n, suggesting no leakage. However, the only condition under which *proc* in fact leaks no information is if I('other') is non-positive or positive with equal probability from the adversary's perspective.

2) An alternative measure: To overcome the limitations of J_n as illustrated above, in this section we propose a leakage measure that is more robust for procedures that employ randomness or inputs in I. For convenience, here we treat all values generated at random within the procedure instead as inputs represented in I; e.g., the first invocation of `rand()` within the procedure is replaced with a reference to, say, I('rand[1]'), the second with I('rand[2]'), and so forth. Intuitively, our measure employs an alternative definition for Y_S that also includes these additional inputs. Specifically, consider the set

$$\hat{X}_{S,S'} = \left\{ \langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \;\middle|\; \langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \in X_S \wedge \langle \mathsf{C}, \mathsf{O} \rangle \in Y_S \cap Y_{S'} \right\}$$

of $\langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle$ triples such that not only is $\langle \mathsf{C}, \mathsf{O} \rangle \in Y_S \cap Y_{S'}$ (c.f., the definition of $J(S, S')$ in (1)), but also the triple is consistent with some $s \in S$ (i.e., $\langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \in X_S$ where $X_S = \bigcup_{s \in S} X_s$). By counting such $\langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle$ triples, the various random values (represented in I) become exposed in $\hat{X}_{S,S'}$ and the number of these values for a given $\langle \mathsf{C}, \mathsf{O} \rangle$ pair act as the "weight" of that pair. We adjust the denominator similarly, resulting in the measure

$$\hat{J}(S, S') = 1 - \frac{\left| \hat{X}_{S,S'} \right|}{\left| X_S \cup X_{S'} \right|}$$

$$\hat{J}_n = \operatorname*{avg}_{\substack{S, S' : |S| = |S'| = n \\ \wedge\, S \cap S' = \emptyset}} \hat{J}(S, S') \tag{8}$$

Note that if $Vars_{\mathsf{I}} = \emptyset$, then $\hat{J}_n = J_n$ since in this case, $\langle \mathsf{C}, \mathsf{O} \rangle \in Y_S$ if and only if $\langle \mathsf{C}, \mathsf{O}, \emptyset \rangle \in X_S$.

The benefit of $\hat{J}_n$ is that it is far less susceptible to the variability that was demonstrated in Sec. III-B1. For example, Fig. 2(c) shows that this measure is stable, independent of M. As we will see in subsequent sections, however, it is also considerably costlier to estimate.

When we use $\hat{J}_n$ in place of J_n, we will annotate measures derived from it using similar notation. For example, $\hat{\eta}^{\min}$ denotes $\eta^{\min}$ computed using $\hat{J}_n$ in place of J_n, and similarly for $\hat{\eta}^{\max}$.

IV. Implementation

In this section, we discuss our implementation for computing the measures discussed in the previous section. Fig. 3 shows the overall workflow for doing so. At the core of our implementation is a hash-based model counting technique that is discussed in Sec. IV-A–IV-C. In Sec. IV-D, we present an adaptation for generating logical postconditions for multiple rounds of procedure executions. In Appendix A, we discuss the use of symbolic execution (e.g., [44], [45]) for generating postconditions, with a focus on a particular optimization that proved useful for our case studies in Sec. VI.

A. Hash-based model counting for J_n

To calculate J_n, we need to estimate $\left| Y_S \cap Y_{S'} \right|$ and $\left| Y_S \cup Y_{S'} \right|$ for randomly selected, disjoint sets S and S' of size n. Since

$$\left| Y_S \cap Y_{S'} \right| = |Y_S| + \left| Y_{S'} \right| - \left| Y_S \cup Y_{S'} \right| \qquad \text{and} \tag{9}$$

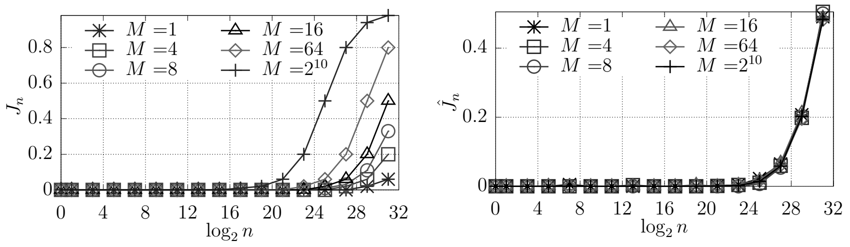
$$\left| Y_S \cup Y_{S'} \right| = \left| Y_{S \cup S'} \right|, \tag{10}$$

to estimate J_n it suffices to estimate $\left| Y_{S''} \right|$ for specified sets S'' (i.e., $S'' = S$, $S'' = S'$, or $S'' = S \cup S'$). In this section, we provide two optimizations for producing such estimates.

1) Estimating $|Y_S|$: Our first optimization is an adaptation of the approximate model counting technique due to Chakraborty et al. [4], which leverages a family of 3-wise independent hash functions to estimate the number $\#F$ of satisfying assignments of a conjunctive-normal-form proposition F of v variables and that runs in fully polynomial time with respect to a SAT oracle. At a high level, this algorithm iteratively selects a random hash function $H^b : \{0,1\}^v \to \{0,1\}^b$ from the family (where b changes per iteration) and a random $p \in \{0,1\}^b$, and computes the satisfying assignments for F for which the hash of the assignment (a string in $\{0,1\}^v$) is p. (Intuitively, this number should be about a $\#F/2^b$.) Through

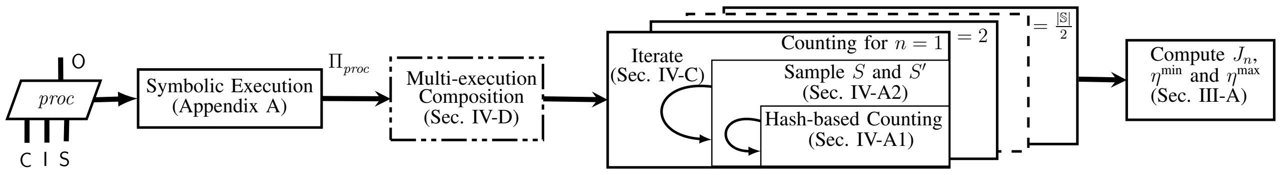
```
proc (C, I, S)
    if (S('secret') = C('test'))
        O('result') ← $rand()$ mod $M$
    else
        O('result') ← $M + (rand()$ mod $16)$
    return O
```

(a) Procedure



(b) J_n for various n and M (c) $\hat{J}_n$ for various n and M

Fig. 2: An example showing limitations of J on procedures with randomness and improvements offered by $\hat{J}$ (see Sec. III-B)



O
proc
C I S
Symbolic Execution (Appendix A)
Π_{proc}
Multi-execution Composition (Sec. IV-D)
Iterate (Sec. IV-C)
Counting for $n = 1$
$= 2$
$= \frac{|\mathbb{S}|}{2}$
Sample S and S' (Sec. IV-A2)
Hash-based Counting (Sec. IV-A1)
Compute J_n, $\eta^{\min}$ and $\eta^{\max}$ (Sec. III-A)

Fig. 3: Workflow of evaluating leakage, from left to right: label the different types of inputs and outputs; generate postconditions Π_{proc} using symbolic execution; optionally, compose multi-execution constraints; perform model counting for different sizes of n; and generate our leakage measures

judicious management of this iterative process, the algorithm arrives at an estimate $\tilde{\#}F$ for $\#F$ that satisfies

$$\mathbb{P}\left((1+\epsilon)^{-1} \cdot \#F \le \tilde{\#}F \le (1+\epsilon) \cdot \#F\right) \ge \delta$$

where error ϵ, $0 < \epsilon \le 1$, and confidence δ, $0 < \delta \le 1$, are parameters and the probability is taken with respect to the random choices of the algorithm.

We estimate $|Y_S|$ similarly, i.e., by iteratively selecting H^b and $p \in \{0,1\}^b$ at random, but apply the hash function only to the C and O values of a satisfying assignment for Π_{proc}. More specifically, we compute the set

$$Z_{S,p} = \left\{\langle \mathsf{C}, \mathsf{O}\rangle \;\middle|\; \langle \mathsf{C}, \mathsf{O}\rangle \in Y_S \wedge H^b(\langle \mathsf{C}, \mathsf{O}\rangle) = p\right\}$$

That is, $Z_{S,p} \subseteq Y_S$ contains the elements of Y_S whose hash is p. Intuitively, this yields an estimate

$$|Y_S| \approx 2^b \cdot \left|Z_{S,p}\right| \tag{11}$$

To reach an estimate of confidence δ, we generate a number of $\langle b, p, \hat{p}\rangle$ triples such that

$$\left|Z_{S,p}\right| \le \alpha \text{ and } \left|Z_{S,\hat{p}}\right| > \alpha \tag{12}$$

where $p \in \{0,1\}^b$, $\hat{p} \in \{0,1\}^{b-1}$, and α is derived from ϵ [4]. Each such triple individually provides an estimate that is within error ϵ with confidence at least 0.78 [4, Lemma 1], and the median of the estimates for all such triples is within error ϵ with confidence that can be increased arbitrarily with more $\langle b, p, \hat{p}\rangle$ such triples. As a special case, if $\left|Z_{S,p}\right| \le \alpha$ at $b = 0$, then $\left|Z_{S,p}\right|$ is an exact count of $|Y_S|$ since $Z_{S,p} = Y_S$.

2) Sampling S, S' of Expected Size n: A second expense of calculating Y_S and $Y_{S'}$ explicitly is in enumerating S and S' themselves, especially if n is large. We can leverage hashing similarly to the method above to avoid enumerating S and S' directly for $n = |\mathbb{S}|/2^b$ for some $b \ge 0$. Specifically, to estimate J_n for $n = |\mathbb{S}|/2^b$, we select H^b and $p \in \{0,1\}^{b-1}$ at random and, for each such selection, define

$$X_p^0 = \left\{\langle \mathsf{C}, \mathsf{O}, \mathsf{I}\rangle \;\middle|\; \exists \mathsf{S} : \Pi_{proc}(\mathsf{C}, \mathsf{O}, \mathsf{I}, \mathsf{S}) \wedge H^b(\mathsf{S}) = p||0\right\}$$
$$X_p^1 = \left\{\langle \mathsf{C}, \mathsf{O}, \mathsf{I}\rangle \;\middle|\; \exists \mathsf{S} : \Pi_{proc}(\mathsf{C}, \mathsf{O}, \mathsf{I}, \mathsf{S}) \wedge H^b(\mathsf{S}) = p||1\right\}$$
$$X_p = \left\{\langle \mathsf{C}, \mathsf{O}, \mathsf{I}\rangle \;\middle|\; \exists \mathsf{S} : \Pi_{proc}(\mathsf{C}, \mathsf{O}, \mathsf{I}, \mathsf{S}) \wedge H^{b-1}(\mathsf{S}) = p\right\}$$

where H^{b-1} denotes the function H^b but dropping the right-most bit from the output. Then, we use the sets

$$Y_p^0 = \left\{\langle \mathsf{C}, \mathsf{O}\rangle \;\middle|\; \exists \mathsf{I} : \langle \mathsf{C}, \mathsf{O}, \mathsf{I}\rangle \in X_p^0\right\}$$
$$Y_p^1 = \left\{\langle \mathsf{C}, \mathsf{O}\rangle \;\middle|\; \exists \mathsf{I} : \langle \mathsf{C}, \mathsf{O}, \mathsf{I}\rangle \in X_p^1\right\}$$
$$Y_p = \left\{\langle \mathsf{C}, \mathsf{O}\rangle \;\middle|\; \exists \mathsf{I} : \langle \mathsf{C}, \mathsf{O}, \mathsf{I}\rangle \in X_p\right\}$$

in place of Y_S, $Y_{S'}$, and $Y_{S \cup S'}$, respectively, to perform the calculations (9)–(10). And, of course, the optimization in Sec. IV-A1 can be used in conjunction with this approach, e.g., computing

$$Z_{p,\hat{p}}^0 = \left\{\langle \mathsf{C}, \mathsf{O}\rangle \;\middle|\; \langle \mathsf{C}, \mathsf{O}\rangle \in Y_p^0 \wedge \hat{H}^{\hat{b}}(\langle \mathsf{C}, \mathsf{O}\rangle) = \hat{p}\right\} \tag{13}$$

for a different, random hash function $\hat{H}^{\hat{b}}$ and random prefix $\hat{p} \in \{0,1\}^{\hat{b}}$. We then use the algorithm summarized in Sec. IV-A1 to estimate $\left|Y_p^0\right|$.

Two more points about this algorithm warrant emphasis:

- Because our algorithm explicitly enumerates the contents of each $Z_{p,\hat{p}}^0$ and $Z_{p,\hat{p}}^1$, when leakage is detected (i.e., $J_n > 0$ for some n) these sets can be used to identify $\langle \mathsf{C}, \mathsf{O}\rangle$ pairs that are in $Y_p^0 \setminus Y_p^1$ or $Y_p^1 \setminus Y_p^0$. These examples can guide developers in understanding the reason for the leakage and in mitigating the problem.
- Because the number of secrets with a random length-b hash prefix p is only of *expected* size $n = |\mathbb{S}|/2^b$, for the rest of

the paper we use a definition of J_n as in (2) but weakened so that $|S|$ and $|S'|$ equal n in expectation.

B. Hash-based model counting for $\hat{J}_n$

The calculations of the previous section require some modifications when we are instead computing $\hat{J}_n$ for $n = |\mathbb{S}|/2^b$. Similar to the previous section, we can use X_p for $p \in \{0,1\}^{b-1}$ in place of $X_S \cup X_{S'} = X_{S \cup S'}$. However, to estimate $\left|\hat{X}_{S,S'}\right|$ for a random S and S', we need a different approach. Specifically, we calculate $\left|\hat{X}_{S,S'}\right|$ by estimating the size of

$$\hat{X}_p = \left\{ \langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \;\middle|\; \begin{array}{ll} \exists \mathsf{S}, \mathsf{S}', \mathsf{I}' : & \Pi_{proc}(\mathsf{C}, \mathsf{O}, \mathsf{I}, \mathsf{S}) \wedge \\ & \Pi_{proc}(\mathsf{C}, \mathsf{O}, \mathsf{I}', \mathsf{S}') \wedge \\ & H^b(\mathsf{S}) = p||0 \wedge \\ & H^b(\mathsf{S}') = p||1 \end{array} \right\}$$

since $\langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \in \hat{X}_p$ iff $\langle \mathsf{C}, \mathsf{O}, \mathsf{I} \rangle \in X_p^0$ and $\langle \mathsf{C}, \mathsf{O} \rangle \in Y_p^0 \cap Y_p^1$. This method does come at considerably greater computational cost, however, due to the duplication of the constraints Π_{proc} in the specification of this set. We will demonstrate this in our case studies in Sec. VI.

C. Parameter settings for computing J_n and $\hat{J}_n$

In the hash-based model counting described above, we use the 3-wise independent hash functions suggested by Chakraborty et al. [4], and due to the large number of XOR clauses in the resulting hash constraints, we use `CryptoMiniSAT 5.0` [46] to enumerate the elements of each $Z_{p,\hat{p}}$. To reduce the complexity of the hash constraints, we concretize their constant bits to minimize the independent support [47] before generating XOR clauses. Multiple estimates of the form in (11), for various values of b (in (11), or respectively $\hat{b}$ in (13)), as prescribed by Chakraborty et al., are used to estimate $|Y_p|$. We parameterized this algorithm with error $\epsilon = 0.45$ and confidence either $\delta = 0.99$ in Sec. V or $\delta = 0.92$ in Sec. VI,[5] for which 50 or 5 $\langle b, p, \hat{p} \rangle$ triples satisfying (12) sufficed, respectively.

We estimate J_n as the sample mean of $J(S, S')$ for sampled pairs S, S' of expected size n (i.e., defined by a $p \in \{0,1\}^{b-1}$ for $n = |\mathbb{S}|/2^b$). For each n we computed J_n using a number of sampled pairs S, S' equal to the larger of 100 and the minimum needed so that the standard error was within 5% of the sample mean. In addition, since J_n is only an estimate and so is subject to error and since that error is influential in the calculation of $\eta^{\max}$ or $\eta^{\min}$ especially when n is small, we round any $J_n \leq 0.025$ down to zero when calculating the measures. $\hat{J}_n$ is computed similarly.

D. Logical Postconditions for Multiple Procedure Executions

In some scenarios it is insightful to observe the behavior of J_n for a procedure *proc* when it is executed multiple times. That is, consider a scenario in which *proc* is executed r times, possibly with relationships among the outputs of one execution and the inputs of another, or simply among the inputs to different executions. Suppose these executions are denoted

$$\begin{aligned} \mathsf{O}_1 &\leftarrow \mathit{proc}(\mathsf{C}_1, \mathsf{I}_1, \mathsf{S}_1) \\ \mathsf{O}_2 &\leftarrow \mathit{proc}(\mathsf{C}_2, \mathsf{I}_2, \mathsf{S}_2) \\ &\dots \\ \mathsf{O}_r &\leftarrow \mathit{proc}(\mathsf{C}_r, \mathsf{I}_r, \mathsf{S}_r) \end{aligned}$$

and that the postcondition of the j-th invocation in isolation is denoted Π^j_{proc} (i.e., Π^j_{proc} is simply Π_{proc} over the variables represented in C_j, I_j, S_j, and O_j). Then the relationships among inputs and outputs can be described using additional, manually constructed constraints $\Gamma^{1\ldots r}_{proc}$. For example, if the secret input to each execution of *proc* is the same, then $\Gamma^{1\ldots r}_{proc}$ would include the statement that 'secret' has the same value in each execution (i.e., $\mathsf{S}_1(\text{'secret'}) = \mathsf{S}_2(\text{'secret'}) = \ldots = \mathsf{S}_r(\text{'secret'})$). Repeating our analysis for the "procedure" represented by the postcondition

$$\left(\bigwedge_{j=1}^{r} \Pi^j_{proc} \right) \wedge \Gamma^{1\ldots r}_{proc}$$

can reveal leakage that increases as the procedure is executed multiple times. We will see an example in Sec. VI.

V. Microbenchmark Evaluation

In this section we evaluate our methodology on artificially small examples to illustrate its features.

A. Leaking more about secret values vs. leaking about more secret values

In Sec. III-A, we showed through an idealized example how a small n is more useful for evaluating the number of secrets about which information leaks, whereas a large n is more useful for evaluating the amount of information leaked about these secrets. Now we will use two simple procedures with a controllable constant M to quantitatively demonstrate the necessity of varying n and the correct usage of $\eta^{\min}$ and $\eta^{\max}$.

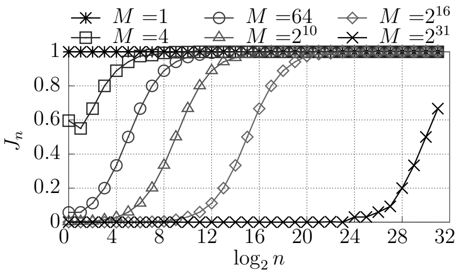
The first procedure, shown in Fig. 4(a), returns the secret value if it is divisible by a constant M and returns zero otherwise, where both $\mathsf{S}(\text{'secret'})$ and M are 32-bit integers. This procedure leaks the same amount of information (the whole secret) about a larger number of secret values if M is decreased. The behavior of J_n shown in Fig. 4(b) is consistent with this observation. Specifically, different values of M induce curves for J_n that differ primarily in the minimum value of n where J_n is large. This behavior is also seen in the value of $\eta^{\min}$ in Fig. 4(c), where $\eta^{\min}$ ranges from $\eta^{\min} \approx 0$ at $M = 2^{31}$ to $\eta^{\min} = 1$ at $M = 1$.

Contrast this case with the procedure shown in Fig. 5(a), which returns the residue class of the secret value modulo a constant value M. As such, as M is increased, more information about each secret is leaked. This is demonstrated in Fig. 5(b), where the curves for different values of M differ in primarily in the maximum value n at which J_n is

[5]The error bound of Chakroborty et al. is conservative; e.g., the results for 95 benchmarks showed less than 5% error in practice even when using $\epsilon = 0.75$ [4].

```
proc (C, I, S)
    if (S('secret') $\bmod M = 0$)
        O('result') $\leftarrow$ S('secret')
    else
        O('result') $\leftarrow 0$
    return O
```

(a) Procedure



(b) J_n for different n and M

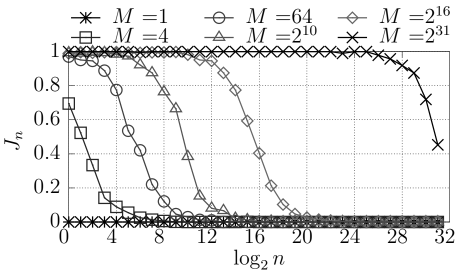
M	$\log_2 \eta^{\text{min}}$	$\log_2 \eta^{\text{max}}$
1	0	0
4	-0.74	0
64	-4.8	0
2^{10}	-8.8	0
2^{16}	-15	0
2^{31}	-30	-0.67

(c) η^{min} and η^{max} for different M

Fig. 4: A procedure that leaks the same amount of information about more secrets as M is decreased (see Sec. V-A)

```
proc (C, I, S)
    O('result') $\leftarrow$ S('secret') $\bmod M$
    return O
```

(a) Procedure



(b) J_n for different n and M

M	$\log_2 \eta^{\text{min}}$	$\log_2 \eta^{\text{max}}$
1	nan	nan
4	-0.6	-30.1
64	-0.0	-25.5
2^{10}	-0.0	-21.8
2^{16}	-0.0	-15.6
2^{31}	0.0	-0.8

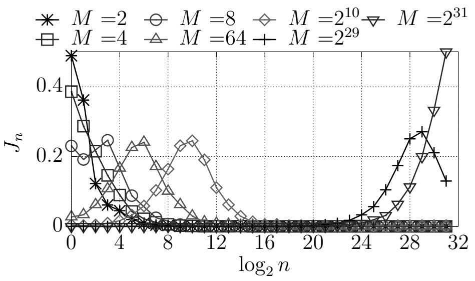
"nan" denotes "not a number," i.e., $\eta^{\text{min}} = 0$ or $\eta^{\text{max}} = 0$

(c) η^{min} and η^{max} for different M

Fig. 5: A procedure that leaks more information about the same secret values as M is increased (see Sec. V-A)

```
proc (C, I, S)
    if (S('secret') $\bmod M$
            $=$ C('test'))
        O('result') $\leftarrow 1$
    else
        O('result') $\leftarrow 0$
    return O
```

(a) Procedure



(b) J_n for different n

M	$\log_2 \eta^{\text{min}}$				$\log_2 \eta^{\text{max}}$			
	$r = 1$	$r = 2$	$r = 4$	$r = 6$	$r = 1$	$r = 2$	$r = 4$	$r = 6$
2	-1.2	-1.1	-1.2	-1.1	-31.4	-31.3	-31.2	-31.3
4	-1.7	-0.9	-0.6	-0.4	-31.0	-30.2	-29.4	-29.0
8	-2.8	-1.7	-0.9	-0.6	-30.6	-29.3	-28.8	-28.3
64	-7.1	-5.4	-3.9	-2.9	-27.1	-25.9	-25.2	-25.1
2^{10}	-11.1	-9.5	-8.2	-7.5	-22.8	-22.0	-21.5	-21.1
2^{29}	-29.9	-28.9	-27.1	-26.5	-3.4	-2.7	-2.1	-1.9
2^{31}	-31.0	-30.2	-28.8	-28.2	-1.2	-0.7	-0.4	-0.2

(c) η^{min} and η^{max} for different M

Fig. 6: Leakage of procedure that checks a guess of secret's residue class modulo M (see Sec. V-A–V-B)

large. Similarly, η^{max} ranges from $\eta^{\text{max}} = 0$ at $M = 1$ to $\eta^{\text{max}} \approx 2^{-0.8} \approx 0.57$ at $M = 2^{31}$.

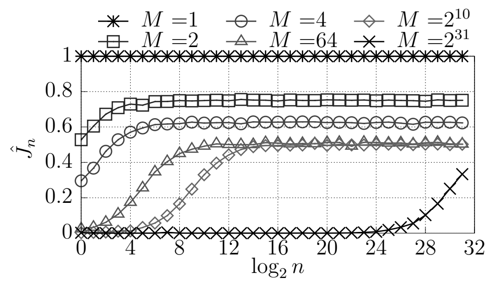
An example that blends these the previous two examples is show in in Fig. 6(a); here the procedure returns 1 if S('secret') $\bmod M =$ C('test') and 0 otherwise, where M is a 32-bit constant. As such, this procedure leaks a lot about a few secret values when M is large, and a little about many secret values when M is small. As shown in the $r = 1$ columns of Fig. 6(c), η^{min} and η^{max} monotonically decreases and increase, respectively, as M grows.

B. Leaking more over multiple rounds

A second way to view the example in Fig. 6 is to consider r procedure executions using the same S('secret') (i.e., S_1('secret') $= \mathsf{S}_2$('secret') $= \ldots = \mathsf{S}_r$('secret')). Our intuition suggests that after $r = M-1$ executions of the procedure, a smart attacker will have learned everything about S('secret') that it can from *proc*; e.g., by setting C_j('test') $= j$, the attacker either will have observed some O_j('result') $= 1$, in which case it knows S('secret') $\bmod M = j$, or else it knows S('secret') $\bmod M = 0$. Consistent with that intuition, in Fig. 6(c), both η^{min} and η^{max} remain steady for $M = 2$ as r increases, since no new information is available to the attacker after $r = 1$. Similarly, for $M = 4$, η^{min} and η^{max} both increase precipitously (by $\geq 74\%$) from $r = 1$ to $r = 2$ and then begin to flatten out (albeit imperfectly—both are estimated values, after all), which is consistent with this intuition that the attacker should learn no new information past $r = 3$. For $M > 4$, each additional procedure execution provides additional information to the attacker about all secrets and

```
proc (C,I,S)
    if (I('rand') mod $M = 0$)
        O('result') ← S('secret')
    else
        O('result') ← I('rand')
    return O
```

(a) Procedure



(b) $\hat{J}_n$ for different n and M

M	$\log_2 \hat{\eta}^{\min}$	$\log_2 \hat{\eta}^{\max}$
1	0.0	0.0
2	−0.9	−0.5
4	−1.8	−0.8
64	−6.5	−1.1
2^{10}	−10.7	−1.2
2^{31}	−31.8	−1.9

(c) $\hat{\eta}^{\min}$ and $\hat{\eta}^{\max}$ for different M

Fig. 7: An example illustrating leakage dependent on randomness (see Sec. V-C)

much more about some (namely those for which it learns the residue class mod M). Correspondingly, both $\eta^{\min}$ and $\eta^{\max}$ increase monotonically along each of these rows.

C. Leaking the secret conditioned on randomness

We now illustrate the ability of our technique to measure leakage from a different randomized procedure from that discussed in Fig. 2. The procedure, shown in Fig. 7(a), returns the secret if a random value is divisible by a constant M and returns that random value otherwise. Clearly, a larger M implies that fewer secret values leak, but those that leak do so completely. This behavior is illustrated by the $\hat{J}_n$ measure shown in Fig. 7(b); the leakage is consistently higher for lower values of M. Similarly, while $\hat{\eta}^{\max}$ remains high for all values of M (never dropping below $\frac{1}{4}$), $\hat{\eta}^{\min}$ ranges from $\hat{\eta}^{\min} = 1$ when all secrets are leaked ($M = 1$) to $\hat{\eta}^{\min} \approx 0$ when few secrets are leaked ($M = 2^{31}$).

VI. Case Studies

In this section, we illustrate our methodology by applying it to real-world codebases susceptible to the inference of search queries via packet-size observations, inference of secret values due to compression results, and inference of TCP sequence numbers. We claim no novelty in identifying these attacks; all are known and explored in other papers, though not in the particular codebases (or codebase versions) that we examine here and typically only through application-specific analysis. Our contribution lies in showing the applications of our methodology to measuring interference in an application-agnostic way and the impact of alternatives for mitigating that interference.

A. Traffic analysis on web applications

Packet sizes are a known side channel for reverse engineering search queries and other web content returned from a server, and defenses against this side channel have been studied using various methods of QIF (e.g., [5], [25], [48]). Specifically, a network attacker can often distinguish between two queries to a web search engine because the response traffic length is dependent on the query. Even packet padding may not hide all secret information [49].

In this section, we use our methodology to analyze the auto-complete feature of search engines to demonstrate our ability to detect the leakage of the user's query from the network packet sizes. Furthermore, we repeat our analysis after applying mitigations suggested in previous work [49]. This allows us to compare the effectiveness of these mitigations to the original implementation.

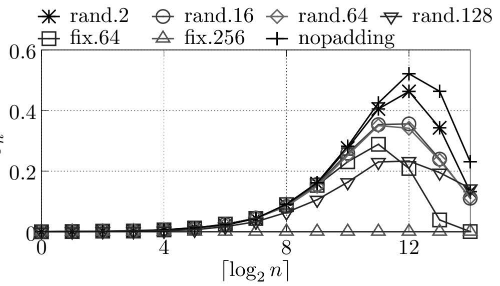
We evaluated a C++ web server called `Sphinx` (http://sphinxsearch.com/), which provides PHP APIs for a client to send a query string to the server. The auto-complete feature then returns a list of keywords that best match the query string. To generate the postcondition that characterizes the auto-complete feature, we marked the query string as the secret (i.e., S('secret') is the query string) and the final application response length as the observable (i.e., $\mathit{Vars}_{\mathsf{O}} = \{\text{'response\_length'}\}$), by injecting only two lines into the server's code. In this application, there was no attacker-controlled input and no other input (i.e., $\mathit{Vars}_{\mathsf{C}} = \mathit{Vars}_{\mathsf{I}} = \emptyset$).

Since the auto-complete results depend on the contents of the server database, we simply instantiated the database with an example containing six keywords and 35 query trigrams (see Fig. 8(a)). When provided an input query string of at least three characters, `Sphinx` returns (content containing) the two keywords with the highest "score" based on matching trigrams in the query string to each keyword's associated trigrams. We also limited queries to three characters drawn from $\{\text{'a'}, \ldots, \text{'z'}\}$ ($\{97, \ldots, 122\}$ in ASCII), yielding $26^3 \approx 2^{14}$ possible queries. Note that instantiating the server with a specific database and limiting the query characters and length as described cannot induce our analysis to provide false positives, though it can contribute false negatives.

We experimented with two types of mitigation strategies. *Random padding* is motivated by protocols like SSH that obfuscate traffic lengths by adding a random amount of padding up to some maximum limit to the application response payload. We experimented with padding lengths of up to 2 bytes ('rand.2'), 16 bytes ('rand.16'), 64 bytes ('rand.64'), and 128 bytes ('rand.128'). *Padding to a fixed length* is a second strategy, which increases the length of the application response payload to the nearest multiple of a fixed length. We experimented with padding to a multiple of 64 bytes ('fixed.64') or a multiple of 256 bytes ('fixed.256'). We "implemented" both of these padding strategies by modifying the postcondition $\Pi_{\texttt{Sphinx}}$ to reflect them (vs. modifying the

Keyword	Trigrams
class	__c _cl cla las ass ss_ s__
code	__c _co cod ode de_ e__
div	__d _di div iv_ v__
the	__t _th the he_ e__
and	__a _an and nd_ d__
title	__t _ti tit itl tle le_ e__

(a) Small database for `Sphinx`



(b) $\hat{J}_n$ for different n

Mitigation	$\log_2 \hat{\eta}^{\min}$	$\log_2 \hat{\eta}^{\max}$
nopadding	-8.3	-1.5
rand.2	-8.4	-1.9
rand.16	-8.5	-2.3
rand.64	-8.5	-2.3
rand.128	-9.0	-2.5
fix.64	-8.5	-3.8
fix.256	nan	nan

"nan" denotes "not a number," i.e., $\hat{\eta}^{\min} = 0$ or $\hat{\eta}^{\max} = 0$

(c) $\hat{\eta}^{\min}$ and $\hat{\eta}^{\max}$ for different mitigations

Fig. 8: Analysis of auto-complete feature of `Sphinx` and mitigation strategies (see Sec. VI-A)

`Sphinx` code directly).

Fig. 8(b) shows $\hat{J}_n$ for the random padding strategies and J_n (which is equivalent to $\hat{J}_n$ since $Vars_{\mathsf{I}} = \emptyset$) for the original, 'fixed.64', and 'fixed.256' strategies. Here, 'nopadding' is the result for original auto-complete in `Sphinx`. In addition, Fig. 8(c) shows the measure $\hat{\eta}^{\min}$ and $\hat{\eta}^{\max}$ for each strategy. Only 'fixed.256' reaches zero leakage, indicated by 'nan' ('not a number'), since any result from `Sphinx` populated with the database in Fig. 8(a) fit within 256 bytes and so resulted in a padded payload of that length. Comparing different padding mechanisms, our measures $\hat{\eta}^{\min}$ and $\hat{\eta}^{\max}$ show results consistent with the intuitive order of the different mitigation strategies in terms of their effectiveness in preventing leakage. Our results suggest that 'nopadding' leaks the most, followed by 'rand.2.' The configuration 'rand.16' was only very slightly worse than 'rand.64', and 'fix.64', which provided similar protection for this setup, and 'rand.128' provided better protection than all others except 'fixed.256.' These results demonstrate the power of our methodology for enabling comparisons of the benefits of different amounts of padding *for this database*. For example, our analysis shows that for this database, 'rand.64' provides little security benefit compared to 'rand.16', despite adding 3× more padding in expectation.

B. CRIME attacks

Our methodology is powerful in accounting for attacker-controlled inputs, and in this section we demonstrate the benefits of this capability by applying it to detect CRIME attacks [6], [50]. A CRIME vulnerability arises when a web client applies "unsafe" compression prior to transmitting a request over TLS. HTTP requests can carry information (e.g., the URL parameters) that an attacker can induce; e.g., if the client visits an attacker-controlled website, then the attacker can induce requests from the client to another, target website with URL parameters that the attacker sets. By observing the lengths of compressed requests to the target website, the attacker can deduce whether the attacker-controlled input shares a substring with a secret contained in the request (e.g., the client's cookie for the target website) that the attacker is unable to observe directly. To be concrete, if the attacker-induced request to the target website is http://target.com?username=*name* then the request will compress better if *name* is a prefix of the client's cookie for target.com.

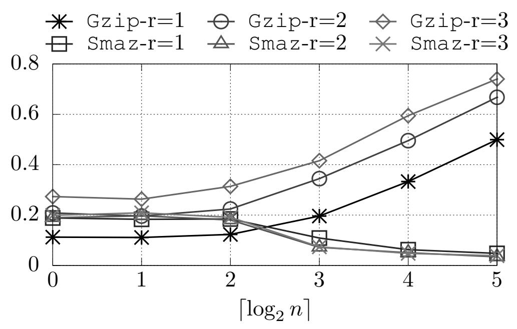
CRIME attacks utilize the property of an adaptive compression algorithm that the encoding dictionary is dependent on both the secret and attacker-controlled variables. As suggested by Alawatugoda et al. [50], a possible mitigation is to separate the compression for the secret and the other parts of the plaintext or to use a fixed-dictionary compression algorithm such as `Smaz` [51]. The latter mitigation, though an improvement, removes the influence of the attacker-controlled input only on the compression dictionary. Consider a two-byte plaintext ab whose first character is secret. If a is '`a`', then this two-byte word will be compressed if b is '`t`' and will be left unchanged if b is '`y`', assuming '`at`' is in the dictionary but '`ay`' is not. Thus, the leakage should not be zero even if a fixed-dictionary algorithm is used.

To analyze this scenario in our framework, we modeled the input for `Gzip` and `Smaz` to be of the form

'`http://target.com/? secret=`' + S('secret') + I('suffix') + '`,username=secret=`' + C('input')

where '+' denotes concatenation. Here, S('secret') and C('input') were each one byte, I('suffix') was two bytes, and the attacker-observable variable was the length of the compressed string. Each byte was allowed to range over 'a', …, 'z' and '0',…,'9'. The S('secret') byte after the first '`secret=`' plays an analogous role to the client cookie in a CRIME attack, i.e., as the secret to be guessed by the attacker, and the '`secret=`' immediately following '`username=`' serves as a prefix to match the first instance of '`secret=`.'

We applied our tool to analyze the leakage susceptibility of `Gzip`-1.2.4 and `Smaz` in this configuration, executed up to three times ($r \in \{1, 2, 3\}$) with the same secret. Our results are shown in Fig. 9. Our results show that for one execution ($r = 1$), `Smaz` is no better than `Gzip`. That is, $\eta^{\max}$ and $\eta^{\max}$ in Fig. 9(b) suggests that `Smaz` leaks less information about some secrets but some information about more secret values versus `Gzip`; as mentioned above, `Smaz` can leak information about a secret value if it composes a word in its dictionary, as well. However, the strength of `Smaz` is revealed as r grows, since its leakage remains unchanged. In contrast, the leakage of `Gzip` grows with r, essentially matching that of `Smaz`



(a) J_n for different n and r

Procedure	$\log_2 \hat{\eta}^{\min}$			$\log_2 \hat{\eta}^{\max}$		
	$r = 1$	$r = 2$	$r = 3$	$r = 1$	$r = 2$	$r = 3$
`Gzip`	-2.04	-1.22	-0.85	-1.00	-0.58	-0.43
`Smaz`	-1.58	-1.55	-1.54	-3.73	-4.02	-3.95

(b) $\hat{\eta}^{\max}$ and $\hat{\eta}^{\max}$ for different r

	C('input')	O('length')	S ('secret')	I('suffix')
`Gzip`	'c'	66	'c'	'oo'
`Smaz`	'r'	36	'f'	'or'

(c) Examples from $Y_S \setminus Y_{S'}$ for samples S, S' ($r = 1$)

Fig. 9: Leakage from `Gzip` and `Smaz` (see Sec. VI-B)

at $r = 2$ and surpassing it at $r = 3$ (in terms of $\eta^{\min}$). This occurs because in each execution of `Gzip`, the attacker has the latitude to select a different value for C('input') and then observe that selection's impact on the length of the compressed string (which in general will change). In contrast, the leakage of `Smaz` is independent of the adversary's choice for C('input'), and so additional executions do not leak any additional information.

As discussed at the end of Sec. IV-A, a side effect of our methodology is identifying some example $\langle \mathsf{C}, \mathsf{O} \rangle$ pairs that lie in $Y_S \setminus Y_{S'}$ or $Y_{S'} \setminus Y_S$ for samples S, S' of secrets, which can help in diagnosing a leak. For example in Fig. 9(c), for `Gzip` in the $r = 1$ case, our tool identified the $\langle \mathsf{C}, \mathsf{O} \rangle$ pair with C('input') = 'c' and O('length') = 66 as being in $Y_S \setminus Y_{S'}$ for a sampled S, S' where $S \ni$ 'c' = S('secret') and I('suffix') = 'oo'.[6] As such, the developer now knows that this $\langle \mathsf{C}, \mathsf{O} \rangle$ pair is consistent with no secret in S'. Similarly, for `Smaz` our tool identified the pair $\langle \mathsf{C}, \mathsf{O} \rangle$ with C('input') = 'r' and O('length') = 36 as being in $Y_S \setminus Y_{S'}$ for a sampled S, S' where $S \ni$ 'f' = S('secret') and I('suffix') = 'or'.

C. Linux TCP sequence number leakage

Known side channels in some TCP implementations leak TCP sequence and acknowledgment numbers [7], [8]. In some cases, these side channels can be used by off-path attackers to terminate or inject malicious payload into connections [8], [52]. The origin of these attacks is shared network counters (e.g., `linux_mib` and `tcp_mib`) that are used to record connection statistics across different connections in the same network namespace.

These counters have been implicated in numerous side channels since version 2.0 of the Linux kernel [53]. For example, the code snippet (without the patch in Lines 6–12) in Fig. 10 leaks the secret `tp->rcv_nxt` in Linux-3.18 TCP. Here, the attacker controls the `skb` input and so the value `TCP_SKB_CB(skb)->seq` that is compared to `tp->rcv_nxt` on Line 5. Based on this comparison, the `NET_INC_STATS_BH` procedure

[6]The output length of 66 exceeds the length of the input string because `Gzip` adds a header to the output. `Smaz` attaches no such header.

```
 1   void tcp_send_dupack(struct sock *sk,
 2                        const struct sk_buff *skb) {
 3     struct tcp_sock *tp = tcp_sk(sk);
 4     if (TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(skb)->seq
 5         && before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt)) {
 6 +     if (before(TCP_SKB_CB(skb)->ack_seq,
 7 +                tp->snd_una - tp->max_window)
 8 +         || after(TCP_SKB_CB(skb)->ack_seq,
 9 +                  tp->snd_nxt)) {
10 +       tcp_send_ack(sk);
11 +       return;
12 +     }
13       NET_INC_STATS_BH(sock_net(sk),
14                        LINUX_MIB_DELAYEDACKLOST);
15       ...
16     }
17     tcp_send_ack(sk);
18   }
```

Fig. 10: A code snippet vulnerable to leaking the TCP sequence number in linux 3.18; lines marked '+' indicate a hypothetical patch with which we experimented (see Sec. VI-C)

increments an attacker-observable counter indicated by `LINUX_MIB_DELAYEDACKLOST` (Lines 13–14). If the attacker can repeatedly cause the procedure in Fig. 10 to be invoked with inputs `skb` of its choice, it can use binary search to infer `tp->rcv_nxt` within 32 executions [8].

The most straightforward mitigation for this leakage is to disable the public counters. This will stop the leakage, but will disable some mechanisms such as audit logging. Another potential mitigation is to increase the difficulty of increasing the public counter, by adding additional checking related to more secret variables. For example, before increasing the `LINUX_MIB_DELAYEDACKLOST` counter, the procedure could also check for correct acknowledgment numbers (`TCP_SKB_CB(skb)->ack_seq` and `tp->snd_nxt`), as shown in the patch in Lines 6–12. As far as we know, our study is the first to compare these potential mitigations for TCP sequence and acknowledgment number leakage.

To analyze the information leakage in this example, we compiled a user-mode Linux kernel [54] as a library. Our target procedure for analysis was `tcp_rcv_established`, which is of the form

```
void tcp_rcv_established(struct sock *sk,
  struct sk_buff *skb,
  const struct tcphdr *th,
  unsigned int len) {
  struct tcp_sock* tp = (struct tcp_sock*) sk;
  ...
}
```

The inputs for `tcp_rcv_established` have many constraints among them when passed in, for instance

$$\texttt{TCP\_SKB\_CB(skb)->seq} < \texttt{TCP\_SKB\_CB(skb)->end\_seq}$$
$$\texttt{tp->rcv\_wnd} \leq \texttt{MAX\_TCP\_WINDOW}$$
$$\texttt{tp->snd\_wnd} \leq \texttt{MAX\_TCP\_WINDOW}$$

To generate constraints for the inputs to `tcp_rcv_established`, we applied symbolic execution to the procedures `fill_packet` and `tcp_init_sock`. Symbolic buffers to represent these inputs and their associated constraints were then assembled within a testing program that called `tcp_rcv_established`. We also stubbed out several procedure calls[7] within `tcp_rcv_established`, causing each to simply return a symbolic buffer so as to avoid symbolically executing it, since doing so introduced problems for KLEE (e.g., dereferencing symbolic pointers).

After generating the postcondition for the procedure `tcp_rcv_established`, we defined the attacker-controlled inputs to be

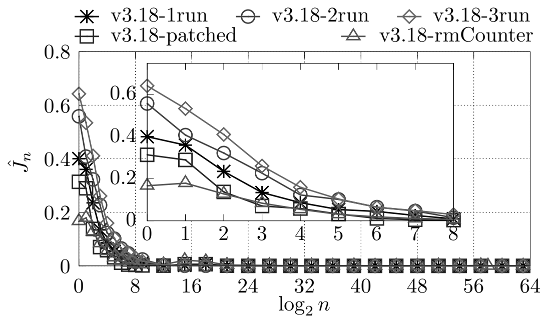
$$\begin{aligned} \mathit{Vars}_{\mathsf{C}} = \{ & \texttt{TCP\_SKB\_CB(skb)->seq}, \\ & \texttt{TCP\_SKB\_CB(skb)->end\_seq}, \\ & \texttt{TCP\_SKB\_CB(skb)->ack\_seq}, \\ & \texttt{tcp\_flag\_word(th)} \} \end{aligned}$$

(each four bytes) and the attacker-observable variables to be $\mathit{Vars}_{\mathsf{O}} = \{$`linux_mib`, `tcp_mib`$\}$. All fields of constrained input structures (e.g., `tp->snd_una` and `tp->max_window`) not covered by $\mathit{Vars}_{\mathsf{C}}$ and $\mathit{Vars}_{\mathsf{O}}$ were added to $\mathit{Vars}_{\mathsf{I}}$, with the secret variables[8] being `tp->rcv_nxt` and `tp->snd_nxt` (each four bytes). We conducted single-execution ($r = 1$, denoted 'v3.18-1run'), two-execution ($r = 2$, denoted 'v3.18-2run') and three-execution ($r = 3$, denoted 'v3.18-3run') leakage analysis. In the multi-execution analysis, we assumed `*sk` to be the same in multiple executions (I_1('`*sk`') $=$ I_2('`*sk`') $= \ldots =$ I_r('`*sk`')) since its fields used in `tcp_rcv_established` would be unchanged or, if changed, would be changed predictably.

The results from this analysis are shown in Fig. 11. The inset graph in Fig. 11(a) is a magnification of the portion of the curve in the interval $[0, 8]$ on the horizontal axis. Specifically, the highest leakage resulted from 'v3.18-3run', followed by 'v3.18-2run' and 'v3.18-1run', as indicated by the $\hat{J}_n$ curves in Fig. 11(a) and the $\hat{\eta}^{\min}$ and $\hat{\eta}^{\max}$ measures in Fig. 11(b). This

[7]Specifically, we stubbed out `get_seconds`, `current_thread_info`, `tcp_options_write`, `tcp_sendmsg`, `prandom_bytes`, `current_thread_info`, `tcp_parse_options`, and `tcp_checksum_complete_user`.

[8]Though we have described our framework so far using one secret variable, it extends trivially to more.



(a) $\hat{J}_n$ per n and version of `tcp_rcv_established`

Version	$\log_2 \hat{\eta}^{\min}$	$\log_2 \hat{\eta}^{\max}$
v3.18-1run	-1.6	-63.0
v3.18-patched	-2.1	-64.1
v3.18-rmCounter	-4.0	-65.6
v3.18-2run	-1.0	-62.1
v3.18-3run	-0.7	-61.6

(b) $\hat{\eta}^{\min}$ and $\hat{\eta}^{\max}$ for versions of `tcp_rcv_established`

Fig. 11: TCP sequence-number leakage (see Sec. VI-C)

shows the potential for the attacker to extract more information about the secrets `tp->rcv_nxt` and `tp->snd_nxt` using multiple executions. This is consistent with the observation that a smart attacker could utilize this side channel to infer one bit per execution [8].

To alleviate this leak, we applied a hypothetical patch shown in Fig. 10 that checks another secret value `tp->snd_nxt` before incrementing the counter for `LINUX_MIB_DELAYEDACKLOST`. Our analysis results (for $r = 1$ execution, denoted 'v3.18-patched') in Fig. 11 shows that the patch alleviated the leakage somewhat. We also tried just deleting Line 5-14 from the original (unpatched) code in Fig. 10. As shown in Fig. 11, this version (denoted 'v3.18-rmCounter') evidently has lower leakage than 'v3.18-patched'. In considering these mitigations, we stress that our patch addressed only the leakage arising from Line 5, and not all sources that leak information about `tp->rcv_nxt` or `tp->snd_nxt` (which are numerous, see Chen et al. [12]). Our results suggest, however, that our methodology could guide developers in mitigating leaks in their code.

D. Performance

Performance of our tool involves two major components, namely the time to compute the postcondition Π_{proc} via symbolic execution, and the time to calculate J_n or $\hat{J}_n$ for different n starting from Π_{proc}. Postcondition generation is not a topic in which we innovate, and so we defer discussion of its costs in our case studies to Appendix A. Here we focus on the costs of calculating J_n or $\hat{J}_n$ for different n starting from Π_{proc}.

Starting from Π_{proc}, the computation of J_n or $\hat{J}_n$ can be parallelized almost arbitrarily. Not only can J_n or $\hat{J}_n$ for each

Sec.	Procedure	$J(S,S')$	$\hat{J}(S,S')$	J_n	$\hat{J}_n$
VI-A	Auto-complete (nopadding)	34ms	56ms	5m	7m
VI-A	Auto-complete (fix.64)	48ms	65ms	6m	8m
VI-A	Auto-complete (fix.256)	43ms	57ms	6m	7m
VI-A	Auto-complete (rand.64)		1.2s		15m
VI-B	`Gzip`		26s		4h
VI-B	`Smaz`		40s		10h
VI-C	v3.18-1run		73s		20h
VI-C	v3.18-patched		67s		20h
VI-C	v3.18-rmCounter		50s		19h

Fig. 12: Average time per estimate ($J(S,S')$ or $\hat{J}(S,S')$) and most expensive overall time (J_n or $\hat{J}_n$) for case studies

n be computed independently, but even for a single value of n, the estimation of $J(S,S')$ or $\hat{J}(S,S')$ can be computed for each pair of sampled sets S, S' and each estimation iteration independently. In Fig. 12, we report the *average* estimation time per sample pair, which indicates that all case studies could finish one estimation in (11) for one sample pair within about one minute. As such, the speed of calculating final pair η^{min} and η^{max} is limited primarily by the number of processors available for the computation.

In our experiments, performed on a DELL PowerEdge R815 server with 2.3GHz AMD Opteron 6376 processors and 128GB memory, we computed J_n or $\hat{J}_n$ per value of n on its own core. As reported in the last two columns of Fig. 12, the time to do so for the *most expensive* value of n ranged from roughly 15m for the auto-complete procedure of Sec. VI-A to about 20h for the Linux TCP implementations of Sec. VI-C. For several of our case studies (see Fig. 12), we experimented with calculating $\hat{J}_n$ even when J_n was sufficient, and found its estimation to cost $\leq 2\times$ that of estimating J_n, due to the duplication of Π_{proc} in $\hat{X}_p$.

To place the above numbers in some context, the $\approx$ 20h (for the worst n, without parallelization) dedicated to computing a value of J_n in the Linux TCP case study of Sec. VI-C involved a procedure $proc$ of which 165 bytes of its inputs were somehow used in the procedure. A naive alternative to our design in which all possible inputs are enumerated and run through the procedure to compute its outputs (and interference measured from these input-output pairs, perhaps as we do) would therefore involve enumerating 2^{1320} possible inputs, which is obviously impractical.

In this light, our technique that performs interference analysis for real codebases in the timeframe of minutes-to-hours (and far faster with parallelization) is a dramatic improvement. Moreover, these results are likely to only improve with advances in symbolic execution and model counting. Even our experimentation with various optimizations for postcondition generation and model counting was not exhaustive. That said, the results above suggest that the costs of our approach are likely to remain sufficiently high for real codebases to preclude its use for interactive analysis by human programmers. Rather, we expect that our analysis could be run as a diagnostic technique overnight, for example.

VII. Discussion and Limitations

Our methodology builds from two tasks that are recognized, difficult challenges in computer science. The first is the construction of a logical postcondition Π_{proc} for a procedure $proc$, for which we leverage symbolic execution. As such, our technique inherits the limitations of existing symbolic execution tools and those incumbent on generating postconditions, more generally. For example, symbolic execution is difficult to scale to some procedures, and challenges involving symbolic pointers and unbounded loops can require workarounds, as they did in our TCP case study (Sec. VI-C). The second challenge problem underpinning our methodology is model counting, which is #P-complete. We are optimistic that future improvements in these areas will be amenable to adoption within our methodology.

Our approach is powerful in that it can be applied to scenarios in which the distributions of inputs—whether they be attacker controlled or other—are unknown, and this is often the case in practice. In some cases, the input distributions are unknowable, especially for $Vars_{\mathsf{C}}$. In others, they may be knowable but require considerable empirical data to estimate (e.g., the distributions of user-input search terms, in a context like that of Sec. VI-A). That said, because it is insensitive to these distributions, it does not offer an immediate way to accommodate these distributions if they are known. Still, our methodology allows these inputs to be accounted for in a principled way, in contrast to others that either disallow them or assign them heuristically.

VIII. Conclusion

In this paper we have suggested a new method for assessing interference and attempts to mitigate it. Informally, noninterference is achieved when the output produced by a procedure in response to an adversary's input is unaffected by secret values that the adversary is not authorized to observe. Following this intuition, we have developed a method to estimate the number of pairs of attacker-controlled inputs and attacker-observable outputs that are possible, conditioned on the secret being limited to a particular sample. The discovery of such pairs that are possible for one sample but not another reveals interference.

We clarified the effectiveness of our strategy both on artificial examples (Sec. V) and on real-world codebases (Sec. VI). Specifically, we evaluated leakage in the `Sphinx` auto-complete feature of its search interface due to its response sizes, and the effectiveness of a variety of mitigations (Sec. VI-A); the CRIME vulnerabilities of adaptive compression in `Gzip` and fixed-dictionary compression in `Smaz` (Sec. VI-B); and leakage of TCP sequence numbers in Linux and the effectiveness of two mitigations of our own design (Sec. VI-C). Within these contexts we also explored leakage over a single procedure execution and over many, and showed that our framework allowed for a useful comparison of how procedures leaked data as the number of executions grows.

Central to our methodology's ability to scale to real codebases is our expression of leakage assessment within a framework that permits the use of approximate model counting (and specifically hash-based model counting). While the resulting tool is not yet quick enough to support interactive use, it is positioned to benefit from advances in symbolic execution and approximate model counting, both active areas of research.

Acknowledgements

This work was supported in part by NSF grants 1330599, 1619450, 1652954, and 1718084.

References

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

[2] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'em, can't live without 'em," in *4th International Conference on Information Systems Security*, 2008, pp. 56–70.

[3] J. A. Goguen and J. Meseguer, "Security policies and security models," in *3rd IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[4] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable approximate model counter," in *Principles and Practice of Constraint Programming*, ser. LNCS, vol. 8124, 2013.

[5] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," in *31st IEEE Symposium on Security and Privacy*, 2010, pp. 191–206.

[6] J. Kelsey, "Compression and information leakage of plaintext," in *9th International Workshop on Fast Software Encryption*, 2002, pp. 263–276.

[7] R. T. Morris, "A weakness in the 4.2BSD Unix TCP/IP software," 1985.

[8] Z. Qian, Z. M. Mao, and T. Xie, "Collaborative TCP sequence number inference attack – how to crack sequence number under a second," in *19th ACM Conference on Computer and Communications Security*, 2012, pp. 593–604.

[9] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *26th ACM Symposium on Principles of Programming Languages*, 1999, pp. 228–241.

[10] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, Jan. 2003.

[11] N. Javanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *27th IEEE Symposium on Security and Privacy*, 2006.

[12] Q. A. Chen, Z. Qian, Y. J. Jia, Y. Shao, and Z. M. Mao, "Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks," in *22nd ACM Conference on Computer and Communications Security*, 2015.

[13] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-site scripting prevention with dynamic data tainting and static analysis," in *14th Network and Distributed System Security Symposium*, 2007.

[14] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *35th ACM Conference on Programming Language Design and Implementation*, 2014, pp. 259–269.

[15] D. E. R. Denning, *Cryptography and Data Security*. Addison-Wesley, 1982.

[16] J. W. Gray, "Toward a mathematical foundation for information flow security," in *12th IEEE Symposium on Research in Security and Privacy*, 1991, pp. 21–34.

[17] D. Clark, S. Hunt, and P. Malacaria, "Quantitative analysis of the leakage of confidential data," *Electronic Notes in Theoretical Computer Science*, vol. 59, no. 3, 2002.

[18] G. Lowe, "Quantifying information flow," in *15th IEEE Workshop on Computer Security Foundations*, 2002.

[19] D. Clark, S. Hunt, and P. Malacaria, "Quantitative information flow, relations and polymorphic types," *Journal of Logic and Computation*, vol. 15, no. 2, 2005.

[20] M. R. Clarkson, A. C. Myers, and F. B. Schneider, "Belief in information flow," in *18th IEEE Workshop on Computer Security Foundations*, 2005, pp. 31–45.

[21] D. Clark, S. Hunt, and P. Malacaria, "A static analysis for quantifying information flow in a simple imperative language," *Journal of Computer Security*, vol. 15, no. 3, pp. 321–371, 2007.

[22] P. Malacaria, "Assessing security threats of looping constructs," in *34th ACM Symposium on Principles of Programming Languages*, 2007, pp. 225–235.

[23] B. Köpf, L. Mauborgne, and M. Ochoa, "Automatic quantification of cache side-channels," in *24th International Conference on Computer Aided Verification*, 2012, pp. 564–580.

[24] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "CacheAudit: A tool for the static analysis of cache side channels," in *22nd USENIX Security Symposium*, 2013, pp. 431–446.

[25] K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen, "Sidebuster: Automated detection and quantification of side-channel leaks in web application development," in *17th ACM Conference on Computer and Communications Security*, 2010, pp. 595–606.

[26] P. Chapman and D. Evans, "Automated black-box detection of side-channel vulnerabilities in web applications," in *18th ACM Conference on Computer and Communications Security*, 2011, pp. 263–274.

[27] Q.-S. Phan and P. Malacaria, "Abstract model counting: A novel approach for quantification of information leaks," in *9th ACM Symposium on Information, Computer and Communications Security*, 2014, pp. 283–292.

[28] P. Mardziel, M. S. Alvim, M. Hicks, and M. R. Clarkson, "Quantifying information flow for dynamic secrets," in *35th IEEE Symposium on Security and Privacy*, 2014, pp. 540–555.

[29] B. Köpf and D. Basin, "An information-theoretic model for adaptive side-channel attacks," in *14th ACM Conference on Computer and Communications Security*, 2007, pp. 286–296.

[30] C. S. Păsăreanu, Q.-S. Phan, and P. Malacaria, "Multi-run side-channel analysis using symbolic execution and max-SMT," in *29th IEEE Computer Security Foundations Symposium*, 2016, pp. 387–400.

[31] Q.-S. Phan, L. Bang, C. S. Păsăreanu, P. Malacaria, and T. Bultan, "Synthesis of adaptive side-channel attacks," in *30th IEEE Computer Security Foundations Symposium*, 2017.

[32] B. Köpf and A. Rybalchenko, "Approximation and randomization for quantitative information-flow analysis," in *23rd IEEE Computer Security Foundations Symposium*, Jul. 2010, pp. 3–14.

[33] A. Biere, M. Heule, and H. van Maaren, *Handbook of Satisfiability*. IOS press, 2009, vol. 185.

[34] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic discovery and quantification of information leaks," in *30th IEEE Symposium on Security and Privacy*, 2009, pp. 141–153.

[35] A. D. Pierro, C. Hankin, and H. Wiklicky, "Approximate non-interference," *Journal of Computer Security*, vol. 12, no. 1, pp. 37–81, Jan. 2004.

[36] R. Ensafi, J. C. Park, D. Kapur, and J. R. Crandall, "Idle port scanning and non-interference analysis of network protocol stacks using model checking," in *19th USENIX Security Symposium*, 2010.

[37] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: From general purpose to a proof of information flow enforcement," in *34th IEEE Symposium on Security and Privacy*, 2013, pp. 415–429.

[38] C. Hritcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. A. de Amorim, and L. Lampropoulos, "Testing noninterference, quickly," in *18th ACM International Conference on Functional Programming*, 2013, pp. 455–468.

[39] F. Dörre and V. Klebanov, "Practical detection of entropy loss in pseudo-random number generators," in *23rd ACM Conference on Computer and Communications Security*, 2016, pp. 678–689.

[40] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, "Decomposition instead of self-composition for proving the absence of timing channels," in *38th ACM Conference on Programming Language Design and Implementation*, 2017, pp. 362–375.

[41] J. Chen, Y. Feng, and I. Dillig, "Precise detection of side-channel vulnerabilities using quantitative cartesian Hoare logic," in *24th ACM Conference on Computer and Communications Security*, 2017, pp. 875–890.

[42] G. Smith, "Quantifying information flow using min-entropy," in *8th International Conference on Quantitative Evaluation of Systems*, Sep. 2011, pp. 159–167.

[43] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith, "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data," *SIAM Journal on Computing*, vol. 38, no. 1, pp. 97–139, 2008.

[44] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2008.

[45] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," in *16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 265–278.

[46] M. Soos, "The CryptoMiniSAT 5 set of solvers at SAT Competition 2016," *SAT COMPETITION 2016*, p. 28, 2016.

[47] A. Ivrii, S. Malik, K. S. Meel, and M. Y. Vardi, "On computing minimal independent support and its applications to sampling and counting," *Constraints*, vol. 21, no. 1, pp. 41–58, Jan. 2016.

[48] G. Kellaris, G. Kollios, K. Nissim, and A. O'Neill, "Generic attacks on secure outsourced databases," in *23rd ACM Conference on Computer and Communications Security*, 2016, pp. 1329–1340.

[49] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton, "Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail," in *33rd IEEE Symposium on Security and Privacy*, 2012, pp. 332–346.

[50] J. Alawatugoda, D. Stebila, and C. Boyd, "Protecting encrypted cookies from compression side-channel attacks," in *Financial Cryptography and Data Security*, 2015, pp. 86–106.

[51] S. Sanfilippo, "Small strings compression library," https://github.com/antirez/smaz, 2009.

[52] Y. Cao, Z. Qian, Z. Wang, T. Dao, S. V. Krishnamurthy, and L. M. Marvel, "Off-path TCP exploits: Global rate limit considered dangerous," in *25th USENIX Security Symposium*, 2016, pp. 209–225.

[53] "Linux blind TCP spoofing vulnerability," http://www.securityfocus.com/bid/580/info, 1999.

[54] J. Dike, "User-mode Linux," in *Annual Linux Showcase & Conference*, 2001.

[55] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *33rd ACM Conference on Programming Language Design and Implementation*, 2012, pp. 193–204.

APPENDIX

A. From Procedure to Logical Postcondition

As mentioned in Sec. III, the logical postcondition Π_{proc} represents the relationship between inputs and outputs induced by procedure $proc$. To extract Π_{proc} from $proc$, we apply symbolic execution to $proc$. After marking each input variable (i.e., each parameter in $Vars_{\mathsf{C}}$, $Vars_{\mathsf{I}}$,[9] and $Vars_{\mathsf{S}}$) symbolic before the user-defined entry point, we utilize `KLEE` [44] or `S2E` [45] to explore all feasible execution paths through $proc$ that reach a `return`. On each path through $proc$, the symbolic execution engine accumulates a set of constraints among symbolic variables implied by the branches taken and assignments computed along that path. These constraints coupled with the assignments for $Vars_{\mathsf{O}}$ defined by our API `make_observable`, as accumulated through the `return` instruction, form the postcondition for the path, and then Π_{proc} is simply the disjunction of the path conditions generated for each execution path.

Symbolic execution can suffer from state explosion, and so we leveraged an optimization in our work to manage this explosion. Specifically, we implemented a searcher to perform *state merging* [55] frequently, wherein the constraints accumulated along two or more execution prefixes ending at the same instruction are disjoined and then simplified to the extent possible (using an SMT solver); execution is then continued from their last instruction, accumulating more constraints into their now-combined constraints. In doing so, these two execution prefixes need only be extended once, versus each being extended separately if no merging occurred.

[9]To model the random input generated from random number generator $rand()$ in symbolic execution, we created a symbolic variable per $rand()$ function call as its returned value.

Sec.	**Procedure**	`KLEE` $\times 1$	`KLEE` $\times 16$	`KLEE` $\times 1$, merging	`S2E` $\times 16$
VI-A	Auto-complete	2d	12h		
VI-B	`Gzip`	3d	21h		8h
VI-B	`Smaz`	2d	18h		6h
VI-C	v3.18	7d	4d	17m	
VI-C	v3.18-patched			18m	
VI-C	v3.18-rmCounter			17m	

Fig. 13: Postcondition generation times for case studies

This optimization dramatically reduced the number of symbolic states managed in one of our case studies in Sec. VI-C, improving the speed of extracting Π_{proc} by more than $600\times$. For this case study, we forced state merging to occur whenever a symbolic state was forked at a symbolic branch. To reduce the complexity of the merged path constraint, however, we avoided merging two path constraints when their expressions for the outputs in O differed or when two path constraints (in conjunctive normal form) had less than half of their conjuncts in common.

A well-known limitation of symbolic execution is how to manage unbounded loops, since these can prevent symbolic execution from terminating. In the case studies of Sec. VI we bounded all inputs, which was enough in these case studies to ensure that symbolic execution terminated. Provided that we bound the input parameters sufficiently loosely to encompass all values they can take on in practice, this bounding does not impact the assessment provided by our measures in practice.

Postcondition generation costs are summarized in Fig. 13. These computations were performed on a DELL PowerEdge R710 server equipped with two 2.67GHz Intel Xeon 5550 processors and 128GB memory. Each processor includes 4 physical cores and had hyperthreading enabled. As indicated in Fig. 13, we experimented with both `KLEE` and `S2E` to generate postconditions, depending on the procedure. In the column headings, a '$\times 1$' or '$\times 16$' indicates the number of processes across which the computation was divided. To enable multi-process support in `KLEE` (i.e., '$\times 16$'), we made a small modification in `KLEE`'s execution engine, to cause it to explore only execution paths starting from a predefined branching prefix. The designation 'merging' indicates the use of the `KLEE` optimization summarized above; as indicated in Fig. 13, this optimization was remarkably effective on the Linux TCP implementations discussed in Sec. VI-C. `S2E` was configured to utilize its concolic execution capabilities.