Securing Mediated Trace Access Using Black-box Permutation Analysis

Prateek Mittal, UIUC
Vern Paxson, UC Berkeley / ICSI
Robin Sommer, ICSI / LBNL
Mark Winterrowd, UC Berkeley

1. INTRODUCTION

The lack of public access to current, real-world datasets significantly hinders the progress of network research as a scientific pursuit. It is often not possible to robustly validate a proposed mechanism, enhancement, or new service without understanding how it will interact with real networks and real users. Yet obtaining the necessary raw measurement data—in particular, packet traces including payload—can prove exceedingly difficult, and not having appropriate traces for a study can stall the most promising research.

There have been extensive efforts by the community at large to change the status quo by providing collections of public network traces. However, the community's major push to encourage institutions to release anonymized data has achieved only very limited success. The risks involved with any release still outweigh the potential benefits in almost all environments. The lack of significant progress in this direction—despite extensive efforts—is an undeniable indication that the community needs a new approach.

An alternative paradigm for enabling network research is *mediated trace analysis*: rather than bringing the data to the experimenter, bring the experiment to the data, i.e., researchers send their analysis programs to data providers who then run them on their behalf and return the output. The community has been using this approach on an *ad hoc* basis for a number of years, but in that form it *fails to scale*: providers only undertake such mediation based on a great deal of trust that the requesting researcher is acting in good faith and that data released via the mediation will not pose any privacy risks.

If as a community we find that for effectively conducting our science we must increasingly rely upon mediated trace analysis, then we must address in a systematic fashion the crucial technical hurdle of ensuring that mediated analysis programs do not leak sensitive information from the data they process. The two frameworks previously proposed for preventing such leaks have the significant limitation of requiring researchers to code their analysis programs in terms of pre-approved modules [6] or a specific language [5]. In this paper we propose a powerful alternative approach that can work with nearly arbitrary analysis programs while imposing only modest requirements on researchers and data providers. The key observation we leverage is that the data provider holds the researcher's program "captive," so to speak: the provider can run it multiple times on different inputs and observe the program's behavior in each case.

Having captive programs creates an opportunity for *permutation analysis*. As a simple example, suppose that a researcher asking for mediated analysis asserts that their program is indifferent to IP addresses—other than that they remain distinct in a one-to-one mapping with end systems—but that in fact the researcher's program searches for the presence of a single particular IP address in a packet trace and flags its presence in a surreptitious fashion in the output. Conceptually, the provider can detect this leakage as follows. They first feed the program the original trace and capture its output. They then permute the trace, consistently altering its embedded addresses, and *diff* the resulting output with that from the first run. If the sensitive address indeed appeared in the original trace but not in the permuted trace, then the outputs will necessarily differ (otherwise, the program failed to leak its presence). If the address did not in fact appear in the original trace, then the outputs may agree (they might not if the permutation happened to accidentally introduce the address), which one might view as "no harm, no foul."

We term such an approach as *black-box permutation analysis*, since it can secure mediated trace analysis without requiring any visibility into the internals of the researcher's program, and thus without imposing any restrictions on how the researcher must code it. However, while the above example is appealing in its conceptual simplicity, applying such analysis in a secure, systematic fashion requires careful consideration of numerous issues. Our work endeavors to illuminate these issues and develop sound approaches for attending to them. In particular, we develop an analytic framework for permutation analysis and employ it to show how to detect violations of a data provider's privacy policy using only a relatively modest number of black-box permutations. We also discuss how our technique can account for innocuous changes in program output via canonicaliza-

tion. Finally, we present experimental results validating the accuracy of our analytic model and the canonicalization mechanism's robustness.

2. RELATED-WORK

For brevity we omit discussion of trace anonymization techniques and known attacks upon some of them as discussed, e.g., in [3, 10, 2]. Mogul and Arlitt first explored mediated trace analysis with the thoughtful *SC2D* design [6]. However, SC2D never proceeded beyond a proof of prototype, leaving many research challenges unaddressed. In particular, its security guarantees depend on the use of pre-approved analysis modules, a specialized language, and independent expert review. Regarding this latter, it is not clear what techniques experts would employ to check for information leaks.

Mirkovic [5] proposed that data providers publish a query language and allow researchers to submit sets of queries expressed using it to run over data. Depending on the provider's privacy policy, the framework allows the use of certain queries on some packet fields in a particular context. The query results however consist of *aggregate information*, such as counts or histograms, imposing a significant limitation on the range of analysis that the system can support.

An alternative approach is to dynamically detect information leaks using *taint tracking* to follow information flow during dynamic execution [8]. However, taint-tracking suffers from both false positives (taint propagates excessively) as well as false negatives (failure to track implicit flows, such as control flow dependencies), and traditionally only checks for a binary attribute (tainted/untainted), which does not fit well for allowing partial information use (declassification of data).

Recent work by Backes *et al.* [1] presents an approach to automatically discover and quantify information leaks. While their assessment procedure is exact, their analysis does not scale to real world programs, since its computational complexity is proportional to the square of program execution paths. We pursue a similar notion but from a different angle, by making probabilistic assertions about information leakage using only a modest number of experiments.

Jung *et al.* [4] present a technique called "differential fuzz testing" in which changes in input are mapped to output changes to infer likely leaks of information. Their system relies on the hypothesis that any output difference generated from two different inputs indicates dependency on a sensitive input parameter, rather than possibly arising (as will often be the case for typical network analysis tools) from benign sources such as the order in which a program reports per-flow statistics.

3. FRAMEWORK

For a given instance of mediated trace analysis we assume that a researcher and a trace provider first negotiate the kind of analysis that the researcher can undertake, to enable the trace provider to evaluate the risks and assess the possible information leaks of concern. We assume that the analysis program runs in a deterministic fashion (if it uses pseudo-random numbers, then it provides a mechanism to set the generator's initial seed), and that the provider runs it in a sandboxed environment that has no Internet connectivity. Thus, the only channel for conveying information back to the researcher is an output file. We assume further that the time scale for verifying that the analysis program does not leak information can range from days to even several weeks, allowing ample time for assessing multiple runs of the program.

We also assume that researchers provide the trace provider with *output templates* that describe the format of the agreed-upon output files, and that the programs generate *audit trails* that define how elements of the input map to positions in the output. These annotations allow the permutation analysis to soundly assess whether a permuted input generates the same output as the original data. We discuss the properties of the output template and audit trail in Section 5. The researchers needn't restrict their analysis as long as they ensure that the final results are free of concern. Provider however can decline to release the output if unconvinced of its safety. Our overarching goal is to reduce how much trust providers require when mediating analysis of their data.

Threat model. The primary threat the framework must address is leakage of information undesired by the trace provider. The framework must protect against both unintentional leakage due to errors and deliberate attempts to exfiltrate data. In practice, providers will likely assign levels of trust to different researchers in an informal fashion, based primarily on reputation, but we can bracket this range by considering two threat models. First, *honest-but-curious* researchers intend to comply with the provider's privacy policy, but may misimplement elements of it, and cannot be prevented from examining the results they get back for artifacts outside the negotiated scope. The *adversarial* model, on the other hand, makes no assumptions about the legitimacy of a researcher's actions. They may include in their analysis surreptitious elements that compromise sensitive provider information.

Policy and risk management. In our framework it is incumbent upon the data provider to determine their *policy* for what information they consider sensitive. Clearly, there is great value in tools for assisting with these determinations, but experience to date has shown that such assessments can be quite subtle [10]. Information correlation from external sources of data also poses a challenge. A significant advantage that our approach offers over trace anonymization in this regard is that the

class of deanonymization attacks is smaller: when releasing anonymized traces, one must determine *at the time of release* the full scope of possible future attacks, while for black box permutation analysis we can prevent any direct leaks *at the time of analysis*.

In general, policies that aim to hide user identity must consider a wide range of trace information, including IP/Ethernet addresses, timestamps, packet header "fingerprints," etc., and providers face a fundamental tradeoff in terms of conservative policies vs. utility for research [10]. Examination of the range of possible policies is beyond the scope of this work. For purposes of illustration, we will mainly consider the notion of sensitive IP addresses.

We can consider a partial ordering of use cases based on risk. For example, for IP addresses we might consider entire addresses as sensitive, or allow prefix-preserving comparisons across addresses. We can keep addresses consistent only within flows, or per-host across flows or not even within flows (addresses in every packet are renumbered). For timing, we can retain global time; relative time since the start of the trace; relative time on a per-flow basis; or relative time consistent for individual hosts (though we must then select which host of the pair in a flow to use).

Our framework can accommodate any of the above policies—the provider expresses which they want to use by selecting the specific mechanisms used to generate permutations of the original trace data. In principle, the privacy policy that drives permutation can be quite expressive; for example, we can restrict the range of permuted values to a specific set of values (an *anonymity set*), a particularly attractive feature for preserving differing classes of machines in the network (e.g., end hosts vs. servers). Privacy policies can also make use of domain information, such as restricting the range of permuted addresses to the data provider's IP address range.

4. ANALYTIC MODEL

A key question for permutation analysis concerns the number of permutation experiments required to achieve a given confidence level about the results. Just running the program on the original trace and a permuted version of it does not suffice to guarantee 0 bits of information leakage. For instance, a program taking a 32-bit IP address as input may have 2 execution paths. Suppose that the program travels the first execution path if the host address ranges between `0/8` and `127/8`, and the second path otherwise. If both the original and permuted values both fall within or outside of `0/8` ... `127/8`, then we will not observe any changes in the program's behavior, despite the program leaking 1 bit of information. In some ways the question of how many permutations are required lies at the heart of the viability of our approach. One can in principle comprehensively discover all information leaks by trying *all* possible input values (thereby exercising all execution paths), but that clearly imposes an intolerable computational burden. In this section we examine the degree to which we can detect leaks in a probabilistic fashion, and develop an analytic model for understanding the confidence level gained by conducting a given number of permutation trials.

Consider the analysis program as a mapping from the input to the output; the output of the program may thus reveal some information about the input. For example, if the program is a one-to-one mapping, the output may reveal *all* information about the input. On the other hand, if the output of the program does not depend on the input at all, then the output reveals no information about the input. In general, a program P induces a set of input equivalence classes $R_1, R_2 \ldots R_N$, such that all inputs within a particular equivalence class R_i yield the same output. Now, an observation o of the program's output reduces the attacker's uncertainty about the value of the input, I, to the members of the corresponding equivalence class, which we denote as $R_{i(o)}$. Let us denote the attacker's uncertainty at this point by $H(I|o)$, which we quantify using the information theoretic metric of entropy. In general, if the attacker knows that some values in the equivalence class are more likely to appear than others, then we have $H(I|o) = -\sum q_j \log_2 q_j$, where q_j is the probability of the j'th element in the equivalence class $R_{i(o)}$ being the value of the input. However, because of the deterministic nature of the program, all members of the equivalence class $R_{i(o)}$ yield the same output o with certainty, and the attacker cannot distinguish amongst the members of $R_{i(o)}$. Thus, their uncertainty about the input is uniform over all elements of $R_{i(o)}$, which gives us: $H(I|o) = \log_2 |R_{i(o)}|$.

We then can express the net information leaked (L) as the change in uncertainty, i.e., $L = H(I|\Phi) - H(I|o) = \log_2 2^b - \log_2 |R_{i(o)}|$, where Φ is simply the null observation and b is the length of the sensitive value in bits. Observe that it is computationally expensive to exactly compute $R_{i(o)}$, since doing so may require up to 2^b permutation experiments. Thus we endeavor to probabilistically estimate the size of $R_{i(o)}$ using only a small number of permutation experiments, m. There is an important tradeoff in the choice of m: increasing it yields better estimates of information leakage, but at the expense of increased processing. At the end of m permutation experiments, suppose we observe x permuted inputs to lie in the equivalence class $R_{i(o)}$. Let X be the random variable indicating the total number of test inputs belonging to equivalence class $R_{i(o)}$, and $Z_{i(o)}$ the probability of a random input falling into $R_{i(o)}$. We wish to estimate $Z_{i(o)}$, since from it we can then compute the size of the equivalence class as $|R_{i(o)}| = Z_{i(o)} \cdot 2^b$. First, we compute the probability of observing a given

size $X = x$, as follows:

$$P(X = x|Z_{i(o)} = z_{i(o)}) = \binom{m}{x} {z_{i(o)}}^x (1 - z_{i(o)})^{m-x} \quad (1)$$

We now use Bayes' rule to compute the probability $Z_{i(o)}$ given the observation x:

$$P(Z_{i(o)} = z_{i(o)}|X = x) = \frac{P(X = x|Z_{i(o)} = z_{i(o)}) \cdot P(Z_{i(o)} = z_{i(o)})}{\sum_{z_{i(o)}} P(X = x|Z_{i(o)} = z_{i(o)})} \quad (2)$$

We can use the prior distribution $P(Z_{i(o)} = z_{i(o)})$ to encode knowledge about information leak: for the honest-but-curious model, we can use a Gaussian centered around the negotiated information leak value, whereas for the adversarial model, a distribution uniform over the range of possible values. We can finally compute $H(I|o)$ as the weighted sum of conditional entropies, giving us a quantitative measure of the attacker's uncertainty after m permutation experiments:

$$H(I|o) = \sum_{z_{i(o)}} P(Z_{i(o)} = z_{i(o)}|X = x) \cdot \log_2(z_{i(o)} \cdot 2^b).$$

Note that so far we have tried to estimate $|R_{i(o)}|$ only from the inputs that fall into that equivalence class. We can improve upon the above uncertainty estimate if we assume that all equivalence classes are of uniform size. We can justify this assumption for both non-malicious adversaries (accidental mistakes) and some malicious adversaries (as this is their best attack strategy, given uniform *a priori* uncertainty about the inputs). To do so, assume that $Z_{i(o)}$ has the form $\frac{1}{N}$, given N equivalence classes. Suppose that at end of m permutation experiments, we observe r distinct outputs with frequency $x_1, x_2, \ldots, x_r$. We can readily generalize Eqn (1) and Eqn (2) to consider these observations as a random *vector* X, gaining additional diagnostic power by using the corresponding multinomial.

Extension to multiple fields. So far we considered only a single sensitive b-bit input. Now suppose we have two sensitive fields of lengths b_1 and b_2. A naive approach might consider this scenario as a single sensitive input of size $b_1 + b_2$, but doing so increases the number of required permutations for accurate results exponentially. However, if the two sensitive fields are independent (for example, IP addresses and packet-capture timestamps), then we can simply apply permutation to one field first, keeping the other fixed, and then vice versa. For the general case of l independent fields, we permute one field at a time, keeping the remaining $l - 1$ fields fixed. In this way, the number of experiments increases only linearly in the number of sensitive fields.

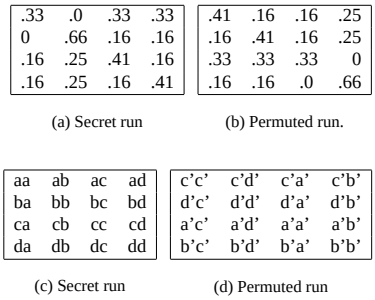
On the other hand, observe that if two fields are *not* independent (e.g., TCP timestamp option and the corresponding echo reply, or IP address and IP checksum), then if the privacy policy specifies one of these, we need to permute the other one accordingly in order to keep the semantics of communication intact. If we are unaware of coupling between two fields, and permute only one of them, then the output may change drastically (due to the violated semantics) even if the program does not leak any information—a failure that is conservative in terms of flagging a possible leak where one does not occur.

Extension to multiple packets. We have framed the discussion in this section in terms of a single sensitive input, such as an address in a given packet header. In practice, traces may of course contain millions of sensitive inputs (e.g., each distinct address), and our techniques need to accommodate such scale. To do so, we introduce the notion of *wholesale permutation*, where in m permutation experiments, we permute each sensitive input m times, and thus the number of permutation experiments does not increase. To illustrate, suppose we have a policy to reveal only the first k bits of information per address. Then we can simply perform m wholesale permutations on the remaining bits, and ensure that outputs reside in the same equivalence class. The problem becomes slightly more complex if the program wishes to use *differing* k bits for each IP, determined dynamically at run time. To handle this case, the program needs to reveal a meta-file specifying which k bits per IP it used, so that we can permute the remaining bits and check for equivalence, as discussed in the next section.

5. EXPERIMENTAL VALIDATION

For a first evaluation in a real-world setting, we assess our framework's performance with three packet analysis tools borrowed from past measurement studies: *(i)* a script that examines TCP sessions for not-yet-acknowledged payload chunks [11], implemented in the language of the Bro network intrusion detection system, which we call *BroAckStat*; *(ii) GraphSplicer* [7], a tool for identifying structures in network communication patterns; and *(iii) TCPTrace*, a tool deriving comprehensive flow-level statistics. We assume the position of a data provider who is handed each of these tools for processing a local packet trace. For our experiments, we consider a simple privacy policy: the analysis tools must not leak *any* information about IP addresses in the trace. Before we can proceed, however, we first need a mechanism for assessing the equivalence of outputs from multiple executions of each tool.

Output Canonicalization. Comparing the outputs from two different executions of a tool is not straightforward. A simple character-level *diff* will typically not suffice, as an output might contain fields that, while depending on the sensitive input, will not be passed on to the researcher. In particular, this will often be the case if the researchers are relying on third-party analysis tools



.33	.0	.33	.33
0	.66	.16	.16
.16	.25	.41	.16
.16	.25	.16	.41

(a) Secret run

.41	.16	.16	.25
.16	.41	.16	.25
.33	.33	.33	0
.16	.16	.0	.66

(b) Permuted run.

aa	ab	ac	ad
ba	bb	bc	bd
ca	cb	cc	cd
da	db	dc	dd

(c) Secret run

c'c'	c'd'	c'a'	c'b'
d'c'	d'd'	d'a'	d'b'
a'c'	a'd'	a'a'	a'b'
b'c'	b'd'	b'a'	b'b'

(d) Permuted run

Figure 1: *GraphSplicer* output (a/b); audit trail (c/d).

producing output they cannot control. *TCPTrace*, for example, outputs IP addresses in clear text as part of a flow identifier. If not part of the output that is shipped back, such fields need to be excluded for output comparison. We note that with some tools innocuous output changes can be very subtle. For example, a change in the computational order might affect just the lower order bits of some output variables, such as when recording timestamps of certain activity. Furthermore, often the *ordering* within an output can change when permuting an input, even though its semantic meaning remains the same (e.g., *BroAckStat*'s output gets rearranged when permuting IP addresses as its order depends on flow identifiers.) Finally, even if a client fully controls the output format (such as for analysis specifically written for the pursued study), it can still be quite challenging to come up with a canonical format suitable for determining the equivalence of two outputs directly. *GraphSplicer*, for example, prints out a matrix of probabilities indexed by host pairs. Permuting input IP addresses changes the order of these matrix entries in non-predictable ways, as depicted in the examples shown in Figure 1(a) and 1(b).

For these reasons, we introduce two transformations to canonicalize a tool's output: *output templates* and *audit trails*. The former describe an output's layout in a simple format we devised, indicating the relevant fields, their expected order, and any additional processing options to apply to them (e.g., truncating lower order bits). Such a template should be provided by the researcher, and we implemented a tool for the provider that canonicalizes an output file accordingly. For those cases where simple post-processing does not suffice (e.g., *GraphSplicer*), we require the analysis to generate an *audit trail* that records meta-information about the processing. The audit trail annotates every output field with a corresponding tuple that specifies its dependencies on sensitive inputs. Figures 1(c) and 1(d) show two such audit trails corresponding to the *GraphSplicer* runs discussed above. The elements in the audit trail comprise of IP address pairs corresponding to matrix elements in the *GraphSplicer* output. Having these trails, we check the isomorphism of the outputs as follows. We simultaneously parse the output generated from the secret input and its audit trail. The first element in the output is ".33", with a dependency in the audit trail of the form `aa`. Our permutation mapping determines the corresponding dependency to look for in the permuted run's audit trail, i.e., `a'a'`. We find it in row 3 / column 3, and match the output element located at that position to the secret run's output. Both output elements are ".33", and thus they match. Similarly, for all other fields, we determine whether the two outputs are indeed isomorphic. A crucial observation here is that audit trails strictly help us with establishing equivalence across runs; any imprecision (or intentional error) in their specifications will lead to overestimates of information leaks rather than underestimates.

Validating canonicalization. To assess the correctness of canonicalization, we wrote output templates for *TCPTrace*, *BroAckStat* and *GraphSplicer*, masking any information about IP addresses. We also modified *GraphSplicer* to leave an audit trail annotating its output. Note that only tens of lines of code were required to modify *GraphSplicer*. With these output templates and audit trails in place, none of the three tools leaks information about IP addresses, and therefore any "permuted outputs" produced from permuted inputs should be isomorphic to the original result. To verify this, we used a public packet trace recorded inside the Lawrence Berkeley National Laboratory (LBNL) [9], consisting of 4169 unique IP addresses and about 2.26 million packets. We ran the analysis tools on this trace, using $m = 50$ permutation rounds. Indeed, for each tool our framework correctly classified all 50 permuted outputs as equivalent to the original ones.

Focusing on *GraphSplicer*, we then instrumented malicious versions of the tool to launch a set of example attacks chosen from four different categories: *(i) Non-targeted attacks revealing complete information:* Print all IP addresses; *(ii) Non-targeted revealing partial information:* For each IP address, print 0 if it lies in `0/8` ... `127/8`, 1 otherwise; *(iii) Targeted revealing complete information*: For the node sending the largest number of packets, print its IP address; and *(iv) Targeted revealing partial information*: For the same node, print 0 if it's inside `0/8` ... `127/8`, 1 otherwise.

Conceptually, for the first three attack classes, we should find no permuted output isomorphic to the original one, considering the large number of IP addresses present in the trace. For the fourth class, we expect that about half of the permuted outputs are isomorphic to the original, since the sensitive range spans half the address space. Next, we ran the malicious versions on the LBNL trace using $m = 50$ permutation rounds. For the first three attack classes, our framework indeed found that all 50 permuted outputs were *not* isomorphic to the original one; while for the fourth class, it determined that 26 were not isomorphic. These results match our intu-

ition and confirm the robustness of the canonicalization.

Validating the analytic model. To assess our analytic model's power, we used the above results with the random variable X set to the number of permuted outputs that are isomorphic to the original. For the honest *GraphSplicer* versions ($X = 50$ and $m = 50$), the model correctly predicts an information leak of exactly 0 bits. For the malicious *GraphSplicer* version, we focus our analysis on the most interesting scenario, the fourth attack class. In this case ($X = 24$, $m = 50$), the model estimates an information leak of 1.0025 bits, which is indeed very close to the true value of 1 bit.

We further examined the model's prediction for a more general targeted attack that leaked k bits of information. We simulated m permutation rounds to estimate the value of X, and then used this as input to the analytic model to predict the expected information leak. For varying values of m, Figure 2 shows the comparison between the actual information leak of a program and the predicted value averaged over 1000 iterations of the simulation. Model 1 and 2 correspond to Section 4's cases of non-uniformly and uniformly sized equivalence classes, respectively. Considering the first model, we see that while with $m = 10$ rounds the predictions are still rather imprecise, $m = 50$ rounds are already sufficient for small values of actual leakage ($0 - 4$ bits). However, for higher degrees of leakage, we have to resort to hundreds of permutations, and even then the error is still quite large for 10 bits. As discussed in Section 4, Model 2 provides additional diagnostic power by considering uniformly sized equivalence classes. In this case, we see that we achieve accurate predictions with much fewer rounds: for an actual leakage of 10 bits, the model predicts 10.2 bits with only a 100 permutations.

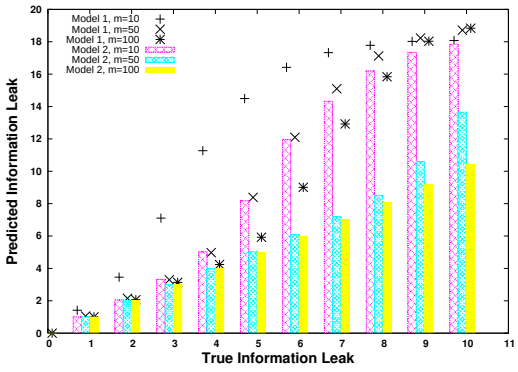


Figure 2: Validation of Analytic Model

Additional Concerns. One final concern regards shipping the output to the researcher. Once black-box permutation analysis determines that the program satisfies the data provider's privacy policy, we take an input that lies in the same equivalence class as the secret input, and ship *its* output to the researcher. This step eliminates any covert channels exploiting the ordering of lines in the case that order cannot be predicted.

6. SUMMARY

We proposed black-box permutation analysis: a powerful approach to securing mediated trace access that can work with nearly arbitrary analysis programs. By permuting sensitive fields in the input trace, and analyzing resulting changes in the program output, we are able to detect information leaks. Black-box permutation accommodates expressive privacy policies, and can detect violations of these policies using only a modest number of permutation rounds, as shown by our analytic model. Our technique can account for innocuous changes in program output via canonicalization using a researcher-supplied *output template* and an *audit trail* generated at run time. Avenues for future work include analyzing security over multiple experimental runs, and helping the trace provider devise appropriate privacy policies.

7. ACKNOWLEDGMENTS

We would like to thank Jelena Mirkovic for sharing her survey of network trace utilization. This work was supported in part by NSF Award CNS-0905631. Opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] M. Backes, B. Köpf, and A. Rybalchenko. Automatic Discovery and Quantification of Information Leaks. In *Proc. IEEE S& P*, 2009.

[2] S. E. Coull, C. V. Wright, F. Monrose, M. P. Collins, and M. K. Reiter. Playing devil's advocate: Inferring sensitive information from anonymized network traces. In *NDSS*, 2007.

[3] J. Fan, J. Xu, M. H. Ammar, and S. B. Moon. Prefix-preserving ip address anonymization: measurement-based security evaluation and a new cryptography-based scheme. *Comput. Netw.*, 46(2):253–272, 2004.

[4] J. Jung, A. Sheth, B. Greenstein, D. Wetherall, G. Maganis, and T. Kohno. Privacy oracle: a system for finding application leaks with black box differential testing. In *Proc. CCS '08*, pages 279–288, 2008.

[5] J. Mirkovic. Privacy-safe network trace sharing via secure queries. In *Proc. NDA '08.*, pages 3–10, 2008.

[6] J. C. Mogul and M. Arlitt. SC2D: An Alternative to Trace Anonymization. In *Proc. ACM MineNet Workshop*, 2006.

[7] S. Nagaraja, N. Borisov, and M. Caesar. Graphsplicer: Detecting p2p topologies in network traffic. Technical Report, University of Illinois.

[8] J. Newsome and D. Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proc. NDSS*, Feb. 2005.

[9] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. A first look at modern enterprise traffic. In *In Proc. IMC*, 2005.

[10] R. Pang, M. Allman, V. Paxson, and J. Lee. The devil and packet trace anonymization. *SIGCOMM Comput. Commun. Rev.*, 36(1):29–38, 2006.

[11] M. Vutukuru, H. Balakrishnan, and V. Paxson. Efficient and Robust TCP Stream Normalization. In *Proc IEEE S & P*, May 2008.