Privately Finding Specifications

Westley Weimer and Nina Mishra

Abstract—Buggy software is a reality and automated techniques for discovering bugs are highly desirable. A specification describes the correct behavior of a program. For example, a file must eventually be closed once it has been opened. Specifications are learned by finding patterns in normal program execution traces versus erroneous ones. With more traces, more specifications can be learned more accurately. By combining traces from multiple parties that possess distinct programs but use a common library, it is possible to obtain sufficiently many traces. However, obtaining traces from competing parties is problematic: By revealing traces, it may be possible to learn that one party writes buggier code than another. We present an algorithm by which mutually distrusting parties can work together to learn program specifications while preserving their privacy. We use a perturbation algorithm to obfuscate individual trace values while still allowing statistical trends to be mined from the data. Despite the noise introduced to safeguard privacy, empirical evidence suggests that our algorithm learns specifications that find 85 percent of the bugs that a no-privacy approach would find.

Index Terms—Specification techniques, software quality, learning, privacy.

1 Introduction

Software bugs are prevalent and testing remains the primary approach for finding software errors. Software testing is difficult and expensive, so techniques to automatically find classes of errors statically [1], [2], [3], [4], [5], [6], [7], [8], [9], [10] or dynamically [11], [12] are gaining popularity. Such tools can typically find some bugs or verify the absence of some mistakes.

These tools require formal specifications or policies about what programs should be doing and the tools report bugs when programs violate those policies. Specifications typically reflect program invariants [7], [13], structural constraints [14], correct API usage [1], [8], security concerns [2], concurrency [4], [15], or general notions of safety [5], [6], [9].

For example, operating systems should ensure that free pages are blanked before allocating them to a new requesting process to avoid leaking sensitive information [16]. Applications that format untrusted data (for example, from the disk or the network) must sanitize data to avoid being compromised via format string vulnerabilities [17], [18]. Operating system trap implementations and device drivers must validate the ranges of user-level and kernel-level pointers before dereferencing them [19]. As a final example, e-commerce and Web applications must carefully handle user input to avoid SQL injection and cross-site scripting attacks [20]. These specifications guard against many different security vulnerabilities (for example, revealing private data, untrusted accesses to databases, and compromises of remote hosts), but they all include rules of the form some event a must occur before some other event b (for example, "proper sanitization" must occur before "a user-supplied string is used").

• *The authors are with the Computer Science Department, University of Virginia, 151 Engineer's Way, PO Box 400740, Charlottesville, VA 22904. E-mail: {weimer, nmishra}@cs.virginia.edu.*

Manuscript received 23 Feb. 2007; revised 30 July 2007; accepted 29 Aug. 2007; published online 12 Sept. 2007.
Recommended for acceptance by P. McDaniel and B. Nuseibeh.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-0078-0207.
Digital Object Identifier no. 10.1109/TSE.2007.70744.

One promising recent approach has been to extract program invariants [13], [21] or full specifications [6], [12], [22], [23], [24], [25] from program sources or traces of program executions. A *trace* is a sequence of events that can be collected statically from the source code or dynamically from instrumented binaries. Unfortunately, the automated specification mining of traces tends to be imprecise in practice. Miners often produce many candidate specifications, only a few (for example, 1 percent to 11 percent [6], [24]) of which are valid. Techniques producing single specifications tend to produce noisy policies (for example, they are too permissive, too strict, or both).

This paper poses the following question: Is it possible to cooperatively learn specifications while preserving the privacy of participants? On the surface, the answer to this question would seem to be "no." After all, participants enjoy perfect privacy when nothing is shared about their code. On the other hand, specifications are optimally learned when everything is shared about the participants' code. Surprisingly, however, a balance between the two extremes can be struck by making a few critical observations. We can learn specifications without exchanging source code or traces. Modern specification miners only require information about the number of times that a function calls another in a normal versus erroneous trace. Our work takes this requirement one step further and shows that even these exact values need not be shared. We describe a method of perturbing these numbers to values that look like random noise. We call these perturbed values a *blurry trace*. The blurring ensures that the participant's privacy is preserved. Despite this strong privacy protection, we show how we can still learn specifications from all of the blurry traces *in aggregate*.

Fig. 1 outlines our technique. Multiple companies have programs that use the same interface or API. Each company obtains traces from its own software and then blurs those traces before publishing them. Aggregate information from all published traces can be used to aid specification mining. Each company expects that its source code and trace information will remain private and each company's

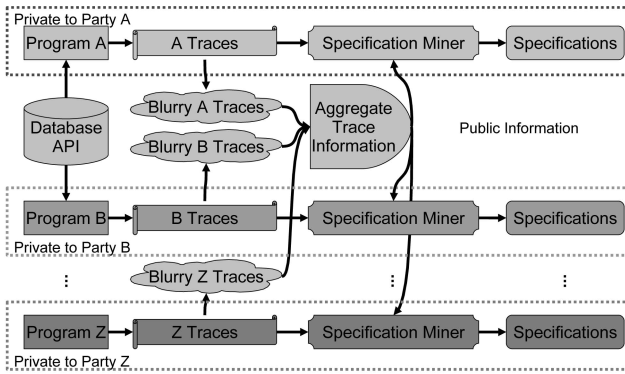

Fig. 1. A diagram showing the privacy-preserving specification model that we propose. The construction of the blurry traces and the use of the aggregate trace information in a specification miner are key concepts in this paper.

specification mining efforts benefit from the aggregate information provided by the other companies.

The main contributions of this paper are:

1. We introduce the notion of privacy to the domain of specification mining. We give a definition of privacy that is motivated by the context of finding specifications. Intuitively, the definition considers the prior knowledge that an attacker has and the posterior knowledge that an attacker gains upon seeing the published information. Privacy is preserved if the posterior knowledge is about the same as the prior knowledge.
2. We introduce blurry traces, which convert information in a trace into essentially random noise. We show that, if a participant publishes a blurry trace, then privacy is preserved.
3. We present an algorithm that allows mutually distrusting participants to work together to mine specifications from these blurry traces. We show that values deduced from the blurry traces approximate their true values, that is, from unblurred traces, provided that there are sufficiently many participants.
4. We also perform an extensive series of experiments. We test our algorithm on almost two million lines of code, mining specifications that find over 700 methods with errors. Despite the blurring, we still find 85 percent of the bugs (and end up with 85 percent of the specifications known) that would be found by no-privacy approaches. Each participant contributes a different illustration of correct usage and, so, the group is able to learn more by collaborating compared to the individuals working alone.

2 Bug Finding and Specification Mining

Automated bug-finding techniques check a program against a specification and report potential bugs where the two differ. If the bug finder is sound, the specification is correct and the model of the program's environment reflects reality, then the bug is real. Otherwise, the bug report is a *false positive*. Bug-finding techniques are often incomplete as well and exhibit *false negatives* as they fail to report some real bugs. The advantage of bug finding is that it can cheaply locate bugs, without the costs of testing. False positives negate that advantage by requiring developers to spend time investigating spurious bug reports.

If the specification used for checking for bugs is incorrect, in general, all of the resulting reports will be false positives. In addition, bugs can only be found with respect to the specifications available. It is thus important to have as many correct specifications as possible.

Given a specification, potential bugs are found using a number of techniques, the most prominent of which are data-flow analysis [3], [5], [6], [9], [10] and iterative abstraction refinement [1], [8], [26]. Different approaches favor different trade-offs with respect to scalability and precision. Almost all existing bug-finding tools can take advantage of the specifications that we mine. A new specification can easily be incorporated into an existing bug-finding system.

2.1 Specification Mining

Specification mining involves extracting models of correct program execution (called *specifications* or *policies* interchangeably) from possibly buggy traces of program behavior. It is conceptually related to time-series prediction, data mining, and some machine learning notions of clustering and classification (for example, [27]). In general, learned specifications can be used for documenting, testing, refactoring, debugging, maintaining, optimizing, and formally verifying programs. Most commonly, however, they are used in conjunction with bug-finding tools to locate program defects.

In practice, specifications take the form of simple finite-state machines[1] over an alphabet of important program events Σ. These *events* are commonly method invocations. Other events are possible (for example, specifications based on variable invariants [12], [22]), but policies based on function call orderings are well established and correspond to intuitive API usage rules. Policies are learned from *traces*, which are sequences of events. Each party collects its own traces statically from the source code or dynamically from instrumented binaries. Traces are often annotated with additional information such as types or error codes. Traces can be interprocedural, covering the program from start to finish, or intraprocedural, listing all of the events that occur within one method invocation.

Intuitively, specification mining might begin by instrumenting a program to print out the name of each function as it is entered. The program is then run on indicative workloads or test cases and the recorded sequences of method invocations become the traces. A mining algorithm interprets those traces and produces candidate specifications: simple finite-state machines that describe legitimate orderings for functions (for example, every `open` must be followed by a `close`).

A formal description of the specification mining problem can be found in [23]. Even restricted versions of the problem are undecidable or difficult, however, and, in practice, all mining algorithms involve approximations. One key practical reason relates to the algorithmic input: the traces. The program under consideration is typically buggy. Since specifications are used to find defects automatically, mining

1. More complicated properties can often be converted to simpler state machines (for example, [28]). In this paper, we use "simple" to refer to two-state DFAs and we concentrate on learning properties of that form.

algorithms and bug finders are usually run on buggy programs rather than correct ones. The traces are thus inherently noisy. Even if `open` followed by `close` is a valid specification, the program may violate it in a few traces or may not exhibit it if there are not enough test cases. Scalable specification mining algorithms typically use statistics and ranking to address this lack of precision: A mining algorithm may produce many candidate specifications, only some of which turn out to be real. For humans, determining if a simple two-state specification is valid is typically much easier than creating a new specification from scratch or debugging an existing one [29] and the final step in the specification mining process involves presenting the candidate specifications to a user for judgment.

2.2 Mining Algorithms

Specification mining is a relatively new research area and only a few prominent techniques are available. Our previous work [24] contains a survey and experimental discussion of the relative bug-finding powers of some miners. We will concentrate on two scalable approaches that attempt to learn pairs of events $\langle a, b\rangle$ corresponding to the two-state FSM specification described by the regular expression $(ab)^*$. Other events unrelated to the specification can occur at any point, so the policy may also be viewed as $(\Sigma^* a \Sigma^* b \Sigma^*)^*$, where Σ is the set of all events.

Finite automata corresponding to $(ab)^*$ might seem too simple to be worthwhile as formal specifications. In fact, 12 simple specifications, such as "not releasing acquired locks, calling blocking operations with interrupts disabled, using freed memory, and dereferencing potentially null pointers," all of which follow the $(ab)^*$ pattern, have been used together to find roughly 1,000 distinct bugs in the Linux and OpenBSD kernel code [30].

We classify a trace as an "error trace" if it terminates with exceptional control flow (for example, a runtime error). Other traces are "normal traces." Let N_{ab} be the number of normal traces that have a followed by b and let N_a be the number of normal traces that have a at all. We define E_{ab} and E_a similarly for error traces. The normal or error nature of a trace can be determined by how it is gathered (for example, using fault injection [31] typically results in error traces) or by what it contains (for example, traces can be inspected for uncaught exceptions or runtime errors). Previous work suggests that programs often fail to adhere to specifications in the presence of runtime errors [10].

Engler et al. [6] propose the ECC technique for mining rules of the form "b must follow a" as part of a larger work on may-must beliefs, bugs, and deviant behavior. They mine event pairs $\langle a, b\rangle$, where a is followed by b in some traces but not in others: $N_a - N_{ab} + E_a - E_{ab} > 0$ and $N_{ab} + E_{ab} > 0$. In addition, a series of data-flow dependency checks is employed to weed out unrelated events (for example, "b must follow a" is more likely to be a real rule if a's return value is one of b's arguments). ECC produces a large number of candidate specifications which are then hierarchically ranked using the z-statistic for proportions. Engler et al. use the ranking because its value grows with the frequency with which the pair is observed together and decreases with the number of counterexamples observed.

In previous work, we proposed the WN technique [24] for mining "b must follow a" rules. Exceptional paths contain more mistakes [10] and we use this trend to learn policies more precisely. WN mines all pairs $\langle a, b\rangle$ such that $E_{ab} > 0$, $E_a - E_{ab} > 0$, and a and b are related by data-flow and program structure requirements. The resulting candidates are ranked in favor of N_{ab}/N_a. These restrictions favor pairs of events that are almost always present on normal paths, are important enough to be enforced on some error paths, but are tricky enough to be forgotten on other error paths.

Together, the ECC and WN mining algorithms have previously found more than 70 specifications that found almost 500 real bugs in almost one million lines of code [24]. Our presentation intentionally casts the ECC and WN miners in a very similar light in order to highlight their use of the same four input values (N_a, N_{ab}, E_a, and E_{ab}), plus some public-type information about the API under consideration. The two mining algorithms have very different false positive rates in practice.

3 Defining Privacy

Our intuitive notion of privacy is that it should be impossible to determine that one participant has buggier traces (and hence buggier software) than another. We will keep *all* trace characteristics, including bug counts, private.

Most participants will not want to make their source code public. Furthermore, program traces are just as sensitive as the source code. Given a specification, each trace can be checked against it for compliance: Having each trace available allows one to count bugs in the original program. Participants cannot suppress all buggy traces, however, because the specifications are not known in advance. The mining goal is to learn *new* policies, some of which will be violated by some input traces. Unless the traces are protected in some manner, successful mining necessarily breaks the privacy by revealing defects in the participants' programs.

3.1 Models of Privacy

One way of attacking the privacy problem is to assume a trusted third party who collects traces from participants and publishes the appropriate specifications. However, in practice, trusted third parties are hard to find. Indeed, it is quite unlikely that a software company will agree to share information about its source code with any other party.

In the absence of a trusted third party, the traditional cryptographic solution is to use secure function evaluation (SFE). In short, SFE gives participants the ability to compute any function of their combined data, without leaking any information beyond the value of the function [32], [33]. The privacy definition is that, with high probability, nothing more is leaked than what could be learned from giving the data to a trusted third party. We discuss the SFE approach more extensively in Section 8.2.

Instead, we follow a model where participants maintain complete control over their data [34]. If a participant wishes to publish perturbed versions of its data, it can do so while simultaneously preserving its privacy. The perturbed versions of the data can be mined either by a miner or collectively by the participants. We go on to prove that large-scale aggregate data can be well estimated, provided there are sufficiently many participants.

3.2 Privacy Definition

We view privacy as a game between an attacker and a participant. The attacker possesses some prior knowledge modeled as a probability distribution D. The participant does not know D. Each participant wants to protect how buggy its code is. Once a specification is released, if an attacker knows how many times b follows a in a participant's traces, then the attacker can deduce a lower bound on how many bugs the participant has and, thus, how much more buggy the participant's code is than the attacker originally believed. We thus seek to prevent an attacker from learning N_{ab}, N_a, E_{ab}, and E_a.

Intuitively, we say that privacy is preserved if the attacker does not learn much from the published information s. Let τ denote a trace and let $f(\tau)$ denote some characteristic of the trace. The functions f that are of interest in this paper are N_{ab}, E_{ab}, N_a, and E_a with respect to the trace τ. One way of defining privacy is to ensure that the attacker's posterior belief that $f(\tau) = x$ is similar to its prior belief. This is precisely the spirit of the first definition of privacy, except that we assume a given set of intervals, denoted $\mathcal{I}$: Privacy is preserved if the posterior belief that $f(\tau) \in I$ is about the same as its prior belief for each $I \in \mathcal{I}$. Note that this definition is more general since it includes, as a special case, the set of intervals where each consists of just one integer value.

Definition 1. *A perturbation algorithm is* ϵ-secure *with respect to* f *if, for any private trace* τ*, it produces a perturbation* $\mathbf{s}$ *such that, for every interval* $I \in \mathcal{I}$,

$$\frac{1}{1+\epsilon} \leq \frac{\Pr(f(\tau) \in I|\text{User published } \mathbf{s})}{\Pr(f(\tau) \in I)} \leq (1+\epsilon),$$

where the prior probability is taken over the attacker's prior beliefs and the posterior probability is taken over the attacker's beliefs and the random coin tosses of the perturbation algorithm.

We now explain why this is a strong definition of privacy. Our perturbation algorithm will produce a blurred version of $f(\tau)$, for example, a blurred version of N_{ab}. Since the specifications that we learn are of the form a precedes b, from this blurred version, it will not be possible to deduce how buggy a single participant's code is: The privacy definition above requires that the attacker learn almost nothing from the published information. Consequently, once a specification of the form b must precede a is mined, an attacker will not be able to determine how many times a participant's code makes the mistake of calling a before b, that is, N_{ab}. Furthermore, it is also not possible for an attacker to determine if one participant has more bugs another. In short, since we will show that the values that a participant publishes approximate random noise, an attacker will not be able to learn anything about a single participant's source code. Nevertheless, with enough participants, specifications can still be discovered.

Although the above definition captures an intuitive definition of privacy, we use a different definition that is easier to work with mathematically. Lemma 1 shows that the definitions are actually equivalent to each other and, consequently, we will actively use the latter definition in our proofs.

Definition 2. *A perturbation algorithm is* ϵ-private *with respect to* f *if, for any private trace* τ*, it produces a perturbation* $\mathbf{s}$ *such that, for any two intervals* I' *and* I'' *in* $\mathcal{I}$,

$$\frac{\Pr(\text{User published } \mathbf{s}|f(\tau) \in I')}{\Pr(\text{User published } \mathbf{s}|f(\tau) \in I'')} \leq (1+\epsilon), \qquad (1)$$

where the probability is taken over the random coin tosses of the perturbation algorithm.

The following lemma states the relationship between the two privacy definitions. The proof is folklore.

Lemma 1. *An* ϵ*-private algorithm is* ϵ*-secure and an* ϵ*-secure algorithm is* 3ϵ*-private.*

In our work, we assume that each participant's trace is independent of every other participant's trace. It seems that this assumption is needed for any input perturbation algorithm because, otherwise, a participant's privacy could be compromised, even if no information is published. In practice, there are likely to be dependencies between participants' data. We do not address this dependency in our work, but note that interparticipant privacy seems to be an open problem in most privacy research to date.

4 Preserving Privacy: Blurry Traces

To mine specifications, we must compute the aggregate values N_a, N_{ab}, E_a, and E_{ab} from the traces. To preserve privacy and utility, we perturb the traces in a manner that allows us to approximately infer aggregate values. The techniques and analysis presented here derive from the work in [35] and [36]. Our contribution is the application and experimental evaluation of these techniques to the specification mining and bug-finding problems.

In practice, there are many relevant events a and b and we need N_a, N_{ab}, E_a, and E_{ab} for all combinations of them. We will explain our handling of N_{ab} for fixed/arbitrary a and b. The other cases are symmetric.

4.1 A Blurry N_{ab}

How can a participant publish a perturbed version of N_{ab}? The main idea involves first representing each participant's data as an indicator vector over the possible values or, more generally, intervals of values that N_{ab} can take. An example is shown in Fig. 2. The participants agree on an upper bound for N_{ab}, in this case 50, and a binning strategy, in this case, each interval has five integer points. In the participant's traces, suppose that b follows a 11 times. Then, the vector $\vec{v}_{ab}$ is an indicator vector for that participant's private data.

If the indicator vector was published, the utility requirement would be met, but the privacy requirement would not be. The participant's value could be pinned down to one of the five numbers. On the other hand, if each element of $\vec{v}_{ab}$ was flipped with probability 1/2, the privacy would be preserved, but nothing could be learned from the resulting totally random vector. Instead, we suggest flipping each element of $\vec{v}_{ab}$ with probability slightly under 1/2, that is, $p = 1/2 - \epsilon$. With this slightly biased flip, we can prove ϵ-privacy.

Bins	0-4	5-9	10-14	15-19	20-24	25-29	30-34	35-39	40-44	45-50
$\vec{v}_{ab}$	0	0	1	0	0	0	0	0	0	0
$\vec{v}'_{ab}$	1	0	1	1	1	0	0	1	1	0

Fig. 2. The top row describes ranges for 10 distinct bins corresponding to the number of times that event a precedes event b in a trace. The middle row is an indicator vector describing a trace where a precedes b 10-14 times. The bottom row is a published perturbed vector, with p close to 1/2.

Lemma 2. *For a trace τ, let $\vec{v}'$ denote the vector obtained by p-perturbing the interval indicator vector $\vec{v}$ for $f(\tau)$. If $p = 1/2 - \epsilon/12$, then $\vec{v}'$ is ϵ-private for f.*

Proof. Letting I and I' be two intervals:

$$\frac{\Pr(\vec{v}' = x | f(\tau) \in I)}{\Pr(\vec{v}' = x | f(\tau) \in I')} \leq \left(\frac{1-p}{p}\right)^2 = \left(\frac{\frac{1}{2} + \frac{\epsilon}{12}}{\frac{1}{2} - \frac{\epsilon}{12}}\right)^2 \leq 1 + \epsilon.$$

□

We assume that all participants perturb with the same probability p. This is a simplifying assumption in that each user could select a different probability p so as to "dial their own" privacy. The bound in the lemma quantifies the resulting mathematical loss in privacy. From a utility perspective, with more values of privacy p, however, more participants will be needed to estimate aggregate data. To simplify the presentation, we assume that p is the same for each participant.

Furthermore, we assume that the perturbation probability p is public information. This is a standard assumption in cryptography (Kerkhoff's principle): Even though the privacy-preserving algorithm is public, an attacker still cannot gain any knowledge about a participant's private data.

4.2 A Blurry Trace

Although the previous analysis assumed that each participant was publishing only one value, specification mining requires an estimate of N_{ab} and E_{ab} for all pairs of functions a and b, as well as N_a and E_a for all a. We advocate that each participant publish a blurry version of each of these values. In other words, given that the participants agree on a binning for each attribute and given that they agree on a perturbation value p, they each produce a p-perturbed version of these values. We call the collective perturbed versions of N_{ab} and E_{ab} for all pairs a, b and N_a and E_a for all a a *blurry trace*.

Note that some unperturbed values are likely to depend on others, that is, the number of times that b follows a and d follows c in a trace may be related. For example, a program with no calls to create new `sockets` will also have no calls to close or write to `sockets`. The dependencies in the unperturbed values potentially create dependencies in the perturbed values, resulting in an increased loss of privacy. We quantify this loss in the following lemma:

Lemma 3. *If a participant releases ℓ perturbed interval indicator vectors $\vec{v}'_1, \ldots, \vec{v}'_\ell$ corresponding to ℓ different functions $f_1, \ldots, f_\ell$, then, for any two possible private values $\langle I_1, \ldots, I_\ell \rangle$ and $\langle I'_1, \ldots, I'_\ell \rangle$,*

$$\left(\frac{p}{1-p}\right)^{2\ell} \leq \frac{\Pr(\wedge_{i=1}^{\ell}(\vec{v}'_i = x_i) | \wedge_{i=1}^{\ell} (f_i(\tau) \in I_i))}{\Pr(\wedge_{i=1}^{\ell}(\vec{v}'_i = x_i) | \wedge_{i=1}^{\ell} (f_i(\tau) \in I'_i))} \leq \left(\frac{1-p}{p}\right)^{2\ell}.$$

The lemma follows since, conditioned on a user's trace, each perturbed vector is generated independently.

The ensuing loss of privacy could, in theory, be very large. If there are thousands of functions, then the number of pairs of functions is on the order of millions. However, this is a decidedly worst-case upper bound on the value ℓ in the previous lemma. Based on our empirical experience, we find that the number of functions that depend on each other is typically at most six (for example, see [12], [22]). If all functions could be partitioned into groups of six so that there are only dependencies within sets of size six, then the loss of privacy can be quantified in the above lemma, with $\ell = 36$. However, even this is a loose upper bound since the proof of the lemma assumes that by "dependency," we mean that the values are identical. Thus, in practice, the loss of privacy will be substantially less than what the preceding lemma suggests.

5 Extracting Aggregate Values

Despite the noise, some aggregate values can be approximately recovered from the perturbed data. For an interval I, let $F_{ab,I}$ be the fraction of participants, with N_{ab} in the interval I. We show how we can estimate $F_{ab,I}$ from the perturbed traces. Let $\hat{F}_{ab,I}$ be the fraction of participants that submit perturbed traces, with a "1" in the interval corresponding to I. Via a simple calculation, we have

$$E(\hat{F}_{ab,I}) = (1-p)F_{ab,I} + p(1 - F_{ab,I}),$$

which implies that

$$F_{ab,I} = \frac{E(\hat{F}_{ab,I}) - p}{1 - 2p}.$$

Thus, if we can estimate $E(\hat{F}_{ab,I})$, then we can estimate $F_{ab,I}$. We next claim that, if we have enough participants, we can, in fact, estimate $E(\hat{F}_{ab,I})$.

Lemma 4. *Assume that we have a collection of $\frac{1}{2\gamma^2} \log \frac{|\mathcal{I}|}{\delta}$ independent participants. Let $\hat{F}^{pub}_{ab,I}$ denote the fraction of participants who publish a perturbed vector, with a "1" in the interval I. With probability at least $1 - \delta$, for every interval $I \in \mathcal{I}$, $|\hat{F}^{pub}_{ab,I} - E(\hat{F}_{ab,I})| \leq \gamma$.*

Proof. By the Chernoff bound, if there are $\frac{1}{2\gamma^2} \log \frac{|\mathcal{I}|}{\delta}$ independent participants, then the probability that $|\hat{F}^{pub}_{ab,I} - E(\hat{F}_{ab,I})|$ is larger than that γ is at most $\delta / |\mathcal{I}|$. Thus, by the Union bound, the probability that any one of the $|\mathcal{I}|$ intervals is not well estimated is at most δ. □

Given estimates for $F_{ab,I}$, we can lower and upper bound $\frac{N_{ab}}{n}$ (where n is the total number of participants) as follows:

$$\sum_{[u,v]\in\mathcal{I}} uF_{ab,[u,v]} \leq \frac{N_{ab}}{n} \leq \sum_{[u,v]\in\mathcal{I}} vF_{ab,[u,v]}.$$

Thus, if we select the midpoint of each interval, then the error in estimating $\frac{N_{ab}}{n}$ is, with high probability, at most the sum over all intervals of the error in $F_{ab,I}$ times half the length of the interval. For example, if each interval has the same length z, then the error is at most $\frac{\gamma}{1-2p}|\mathcal{I}|\frac{z}{2}$.

5.1 Privacy versus Utility Trade-Off

These results illustrate an inherent classic trade-off between privacy and utility. For example, the closer p is to 1/2, the more privacy is given to the participants, but the worse the error in estimating aggregate data such as N_{ab}. As another example, the more intervals used, the more privacy is given to the participants, but the worse the error in estimating large-scale data. The reason is that privacy for a fine-grained set of intervals implies privacy for a coarser grained set. Specifically, privacy for the intervals in $\mathcal{I}$ implies privacy for unions of intervals in $\mathcal{I}$ (proven in Lemma 5). The implication in the other direction does not hold, that is, privacy over a coarse-grained set of intervals does not imply privacy over a finer grain. The solution, however, is not to simply increase the number of intervals since, then, more participants are needed to accurately estimate $F_{ab,I}$. Furthermore, the approximation error in estimating N_{ab} declines. An appropriate balance must be struck between the number/length of intervals and p so as to achieve the desired level of privacy and utility.

We conclude the section with a proof that privacy for the intervals in $\mathcal{I}$ implies privacy for unions of intervals in $\mathcal{I}$.

Lemma 5. *A perturbation that is ϵ-secure for all intervals in $\mathcal{I}$ is also ϵ-secure for all unions of intervals in $\mathcal{I}$.*

Proof. We do the proof for two consecutive intervals and a similar argument works for more intervals. Let $[u, v]$ and $[v, w]$ each be ϵ-secure. We want to show that, then, $[u, w]$ is also ϵ-secure. Let $\alpha = \Pr(f(\tau) \in [u, v]|\mathbf{s})$, $\beta = \Pr(f(\tau) \in [u, v])$, $\gamma = \Pr(f(\tau) \in [v, w]|\mathbf{s})$, and $\delta = \Pr(f(\tau) \in [v, w])$. By assumption, $\frac{1}{1+\epsilon} \leq \frac{\alpha}{\beta} \leq (1+\epsilon)$, and $\frac{1}{1+\epsilon} \leq \frac{\gamma}{\delta} \leq (1+\epsilon)$. We want to show that

$$\frac{1}{1+\epsilon} \leq \frac{\Pr(f(\tau) \in [u,w]|\mathbf{s})}{\Pr(f(\tau) \in [u,w])} \leq (1+\epsilon).$$

Since the probability that $f(\tau) \in [u, w]$ is comprised of disjoint events $f(\tau) \in [u, v]$ or $f(\tau) \in [v, w]$, we can write $\Pr(f(\tau) \in [u, w]|\mathbf{s})$ as $\alpha + \gamma$ and $\Pr(f(\tau) \in [u, w])$ as $\beta + \delta$. We want to show that $\frac{1}{1+\epsilon} \leq \frac{\alpha+\gamma}{\beta+\delta} \leq 1+\epsilon$. Using the assumption, we have that $\frac{\alpha}{\beta+\delta} \leq \frac{\beta(1+\epsilon)}{\beta+\delta}$ and $\frac{\gamma}{\beta+\delta} \leq \frac{\delta(1+\epsilon)}{\beta+\delta}$ so that the sum $\frac{\alpha+\gamma}{\beta+\delta} \leq (1+\epsilon)$, as desired. Similarly, it can be shown that $\frac{\alpha+\gamma}{\beta+\delta} \geq \frac{1}{1+\epsilon}$. □

6 Privacy-Preserving Specification Mining Algorithm

Our algorithm works as follows:

1. A number of mutually distrusting participants agree to mine specifications with respect to an API or library known to all of them. This choice fixes the set of interesting events Σ, from which the various as and bs are drawn.
2. Each participant chooses an interval size S, a number of intervals I, and a perturbing probability p such that that its desired level of privacy is achieved.
3. Global values S_g, I_g, and p_g are set to the largest proposed p and S values and the smallest proposed I value. This satisfies all privacy guarantees.
4. Each participant privately examines its own traces and computes real values of N_a, N_{ab}, E_a, and E_{ab} for all $a \in \Sigma$ and $b \in \Sigma$.
5. Each participant encodes its own N_a, N_{ab}, E_a, and E_{ab} values into indicator vectors with sizes S_g and I_g and then p_g-perturbs[2] and publishes each result.
6. Each participant uses the techniques in Section 5 to recover approximate aggregate values for N_a, N_{ab}, E_a, and E_{ab} that describe the entire cumulative set of traces used by all participants.
7. Each participant mines specifications from that aggregate data.

In Step 1, the participants must all have programs that use a common library or API. Programs with nothing in common cannot benefit from our technique.

In Steps 2 and 3, the participants trade off privacy for utility: preserving privacy sacrifices utility. Our experiments in Section 7 show that generous levels of privacy still allow useful results to be obtained from the perturbed data.

In Step 5, note that the perturbed indicator vectors need not be published simultaneously in a tamper-proof manner. The perturbed vector preserves privacy, regardless of who views it, and participants can post vectors at different times.

The novelty of the algorithm lies in publishing the perturbed data to preserve privacy while maintaining utility: The actual specification mining is performed by an off-the-shelf algorithm on the reconstructed aggregate data. We view this as an advantage: As presented, our algorithm works with both the WN and ECC mining techniques. Other mining algorithms (for example, [25]) could use this framework by extending it to publish perturbed versions of the data that they require.

The mining actually happens after the perturbed traces have been published. The noisy traces serve as input to the mining algorithm (see Section 2.2). Note that the summed values (for example, of N_a) for all participants are added and not averaged: If one participant uses the common API more than others do, thus making a very large N_{ab}, we can still learn the specification $\langle a, b\rangle$ even if other small participants have low values for N_{ab}.

2. In Steps 2, 3, and 5, the participants can choose separate S, I, and p values for each vector. For example, the perturbed vector for N_{ab} might use fewer bins than the one for E_{cd}. This allows participants to fine-tune their privacy requirements on a per-value basis (for example, to deal with values that are not independent, as in Section 4.2).

Program	LOC	Traces	Description
1. axion	65k	8513	database
2. hibernate2	57k	16266	object persistence
3. hsqldb	65k	14381	database
4. openreports	15k	5020	web reporting
5. ireport	149k	8649	visual reporting
6. scheduler	79k	24808	job scheduler
7. squirrel	197k	33254	database client
8. opentaps	266k	55607	business
9. dbmt	4k	3563	data migration
10. jbpm	75k	9780	business
11. jboss	107k	23121	middleware
12. neogia	614k	126384	business
13. cayenne	86k	14102	object framework
14. mckoisql	116k	19651	database
Grand Total	1895k	363099	

Fig. 3. Programs used in experiments, with lines-of-code counts and numbers of associated traces. The programs are presented in the random order chosen for the primary trials (for example, an experiment with three participants involves axion, hibernate2, and hsqldb).

The miner produces candidate specifications (that is, candidate $\langle a, b \rangle$ pairs). Since all of the perturbed traces have been published, each participant can run the miner and get the same results. There is no trusted third party that "does the mining." We view this as an advantage.

7 Experiments

We performed an extensive series of experiments to test the hypothesis that our privacy-preserving technique can produce useful results, even as it safeguards privacy.

Our experiments involved 14 open source programs, totaling 1.9 million lines of code (see Fig. 3). We generated a set of traces for each program. In our experiments, each program represents a separate participant that wishes to learn, without sacrificing its privacy, specifications about the Java Standard Library. In our experiments, the participants always join in the random order in which they are presented in Fig. 3. For example, an experiment with two participants involves axion and hibernate2.

We chose open source programs and APIs because of their availability. In practice, the Java Standard Library is well understood, which allowed us to quickly verify or reject candidate specifications.

7.1 Additional Traces

Our first experiment provides a baseline for our general claim that additional traces increase the precision of specification mining. An abundance of traces mitigates the effects of noisy traces and unindicative traces. In our first experiment, we chose a single program, neogia, for which we had an unusually large number of traces (seven times more than our per-program average).

Fig. 4 shows that increasing the number of available traces increases the precision of the WN specification miner and the number of specifications found. On each input set of traces, the miner outputs scores of candidate specifications. Developers rarely spend effort on inspecting all candidate traces, so we considered only the 20 highest ranked candidates for each trial and manually determined if they were real or false positives. Given more and better traces, automated specification miners produce more and better specifications, which can thus find more bugs.

7.2 Specification Mining

We applied our algorithm in Section 6 to the participants in Fig. 3. Each participant published a set of blurry traces. We applied the WN and ECC algorithms to those blurry traces, collected the resulting candidate specifications, and then manually determined if those candidates were valid policies or false positives.

When perturbing the data, we fixed the number of intervals at $I = 100$. The interval size varied over the value being encoded but was generally $S = 2,000$. Fixing these parameters allowed us to consider the perturbing

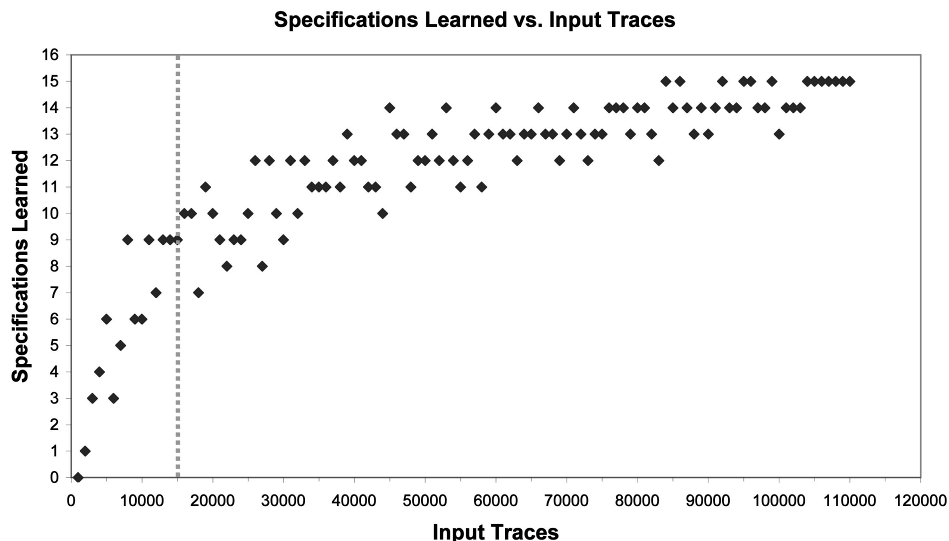

Fig. 4. A plot of the number of real specifications among the first 20 candidates specifications produced by the WN specification miner versus the number of input traces. All traces are from the neogia program. Each point represents an independent random subset of the traces. The dashed line near 15,000 indicates the median number of traces available over the programs in our data set. Note that performance continues to improve with additional traces beyond 15,000: This provides a strong motivation for sharing traces.

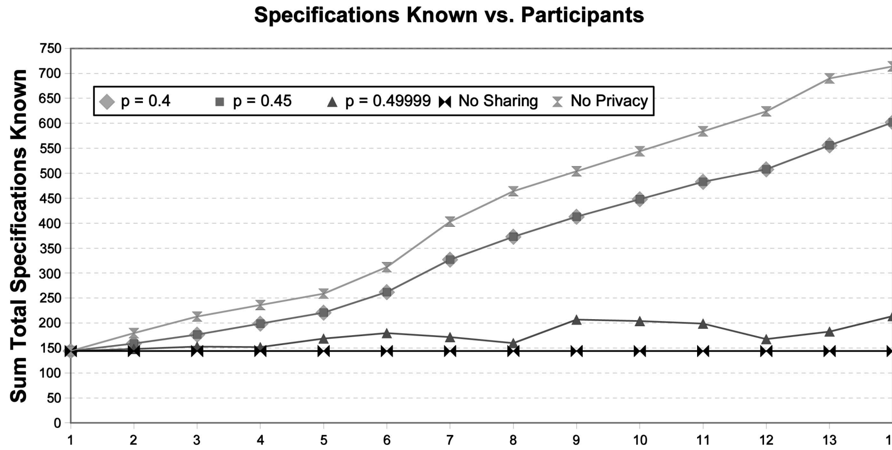

Fig. 5. A plot of the total number of valid specifications known, summed over all 14 potential participants. The horizontal line at 144 indicates the total number of specifications known if each participant works independently, without sharing. Finding 714 specifications is a potential upper bound for this experiment. The $p = 0.45$ curve reaches 85 percent of that with all 14 participants.

probability p as the only independent variable: Higher p values yield more privacy and less utility.

Over the course of all of the experiments presented here, the WN mining algorithm generated 3,658 false-positive specifications (for example, `java.util.Set.add` should be followed by `java.util.Timer.cancel`) and 112 real specifications (for example, `java.sql. Connection. createStatement` should be followed by `java.sql. Statement.close`). High false-positive rates are common in specification mining [6], [10], [29].

We are also interested in the total number of specifications known to all development teams. Beyond an immediate use in bug finding, specifications can also be used to guide development and aid in program understanding. In this experiment, we sum over all 14 potential participants the number of specifications known to that participant. This is different from measuring the total number of unique specifications known anywhere. From a software engineering perspective, we view it as twice as good if two separate participants are both aware of the same valid specification. In such a case, both independent development groups can make local use of the same specification.

Fig. 5 shows a plot of the sum of the total number of specifications known as a function of the number of participants. If no sharing takes place at all and each participant works alone, there are 144 total known specifications: `axion` learns 8, `hibernate2` learns 13, the others learn 123 total, and, on average, each participant learns 10.2 valid specifications. The total knowledge increases as more participants share. For example, with two participants, the aggregate perturbed traces of `axion` and `hibernate2` can be mined to learn 18 valid traces. The number of known traces increases to 159 (18 for `axion`, 18 for `hibernate`, and the same 123 for the nonparticipating others). When all 14 participants work together, they are each able to take advantage of the 43 resulting valid specifications. This is four times, on average, of what they would have when working alone.

Over the course of all of the experiments that we conducted, 51 of the 112 valid specifications mined dealt with the Java Standard Library. Thus, $714\ (= 51 \times 14)$ represents an upper bound of sorts for this mining technique. The true upper bound is not known: No exhaustive analysis of the Java Standard Library has been performed and, in general, it is impossible to know if all potential specifications have been identified (see Section 8.1.2). However, if everything had gone perfectly and all 14 participants had shared all of their data without any concern for privacy, it would have been possible to learn those 51 specifications. In practice, with $p = 0.45$ and 14 participants, we learned 85 percent of that total.

The $p = 0.49999$ curve shows the results of specification mining on very blurry traces. The resulting interval vectors are very close to random bits.

All of the numbers presented above are the result of applying the WN algorithm to the blurry traces. We also applied the ECC algorithm to the traces, but it produced a large number of false-positive specifications. For example, with $p = 0.4$ and only two participants (`axion` and `hibernate2`), ECC reported 36,148 candidate specifications, of which 3,768 had a positive z-statistic ranking. We did not have time to manually verify all of ECC's candidate specifications. We will return to it in Section 7.4.

For the experiments shown in Fig. 5, the runtime cost of preserving privacy is quite low. On average, it took 0.823 seconds per 1,000 traces to construct and populate the interval vectors and another 0.559 seconds per 1,000 traces to perturb the interval vectors on a 3.6 GHz Pentium 4. Preserving privacy on the final experiment, with all 14 participants, took a total of 505 s. However, the blurring can and should be done in parallel, with each participant constructing its own perturbed interval vector. The average time per participant was 36 s, with `neogia` taking the longest at 176 s. These numbers are for $I = 100$ and $p = 0.4$. The time taken scales linearly with I and is not affected by changes in p.

7.3 Bug Finding

One important measure of the utility of a specification is the number of program bugs that it allows you to find. We gathered all of the valid specifications learned via the previous experimental setup. We then checked the source code of the participants against those valid specifications by

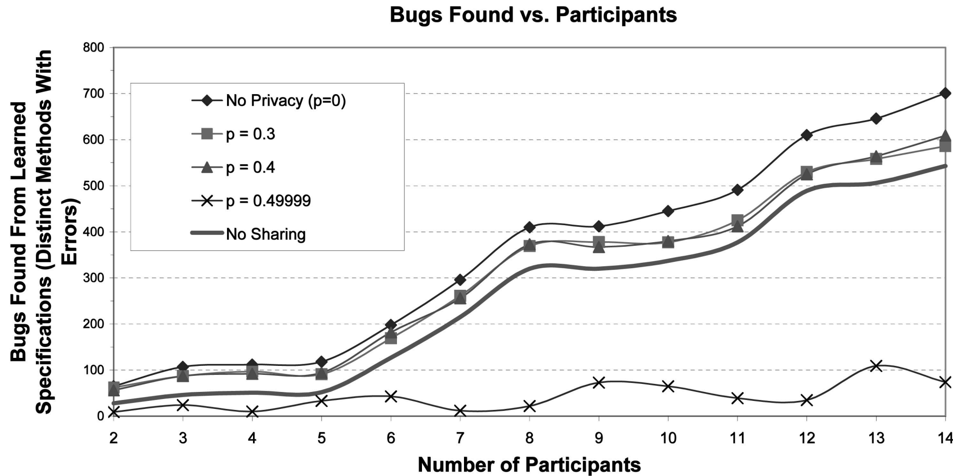

Fig. 6. A plot of the number of bugs found using the specifications learned as a function of the number of participants. Each point is an independent random trial. A method with multiple errors counts as one bug. The $p = 0$ curve shows the utility obtained when privacy is completely sacrificed (all data is published unperturbed). The "No Sharing" curve shows the total number of bugs that the participants find by working alone on their own traces. However, note that this cumulative "No Sharing" curve is hypothetical since it involves participants sharing their learned specifications and bugs without regard to privacy. The $p = 0.4$ curve is, on average, 85 percent of the $p = 0$ curve.

using a standard type-state flow analysis [5], [10]. We counted the number of distinct methods that contained at least one indicated error and called each such method a single bug.

Fig. 6 shows the number of bugs found as a function of the number of participants. The $p = 0$ curve shows the no-privacy result when all values are published unperturbed. This requires mutually trusting parties and represents the highest achievable utility. The "No Sharing" curve shows the total number of bugs participants find by working alone on their own traces [10]. The $p = 0.3$ and $p = 0.4$ curves show that, even with a high degree of privacy for participants, our technique allows useful information to be extracted from the aggregate perturbed values: We mine real specifications that find bugs. The $p = 0.4$ curve is, on average, 85 percent of the nonprivate $p = 0$ curve and is 12 percent better than the "No Sharing" total.

The $p = 0.49999$ curve shows the results of bug finding using only specifications mined from highly perturbed data. Some valid bugs are found, but the participants are better off working alone than using aggregate data that resembles random noise.

7.4 The Price of Privacy

One potential concern is that maintaining adequate privacy would result in traces that were too blurry to be useful. In our experiments, that was not the case.

Miners typically have poor false-positive rates, with 1 percent to 11 percent of candidate specifications turning out to be real [6], [24] in no-sharing, no-privacy scenarios. With $p = 0.4$ and sharing, the average precision was 13.6 percent, which is a minor improvement over previous results [24] or our average $p = 0$ precision of 8.36 percent for this set of programs. Sharing blurry traces does not impair precision, but mining algorithms still require a human expert to evaluate their output.

Beyond precision, another way of measuring the impact of blurry traces on miners involves their ranking of candidate specifications. Developers typically stop examining candidate specifications after a certain point, so if valid specifications have moved down the ranked list, then more developer effort will be required to find and take advantage of them. With two participants, ECC finds a valid specification involving `PreparedStatement.executequery` and `ResultSet.close`. At $p = 0$, it occurs at position 48 on the list, so developers are likely to find and use it. With $p = 0.4$ and blurry traces, it occurs at position 2999.

This loss of ranking order for valid specifications is significant for ECC, but it is mitigated by using a mining algorithm that does not produce as many false positive specifications. For example, for that same two-participant data set, between $p = 0$ and $p = 0.4$ the average valid candidate specification produced by the WN miner moves down 13 percent in the ranked output. However, since WN produces a total of only 78 candidate specifications that must be manually examined, that translates into only an extra 10 candidate specifications to examine. Thus, reasonable privacy can be attained without sacrificing precision or adding high manual verification costs.

7.5 Experimental Conclusions

These experiments support the claim that useful results can be obtained from the aggregate perturbed data produced by our algorithm. Even with high privacy constraints such as p values of 0.4 or 0.45 that make any particular published interval vector similar to random noise, the aggregate information still allows many useful specifications to be learned. While preserving privacy, we still find 85 percent of the bugs (and end up with 85 percent of the specifications known) that could be found by forsaking privacy entirely.

8 Discussion

In this section, we discuss several issues related to the general problem of specification mining. We also compare and contrast our privacy-preserving solution to several simpler techniques.

8.1 Specification Mining Difficulties

Specification mining is complicated by the application-specific nature of important behavior, the difficulty of extracting specifications from documentation, the shortage of useful program traces, and the lack of privacy involved in sharing such traces. All of these issues help motivate the algorithm presented in this paper.

8.1.1 Particular Applications

A specification must be related to the target program in order to be useful: A specification for handling 3D graphics resources is not directly relevant to a user-level Web server. Unfortunately, specifications are difficult to create and debug manually [29]. As a result, frequently used bug-finding tools either use general low-level specifications that are applicable to all programs (for example, "do not dereference a function argument that is known to be a null pointer" [5], [6], [9]) or restrict themselves to a small application domain for which key specifications can be manually gathered over time (for example, handling asynchronous I/O request packets in Windows device drivers [1]). Bug-finding tools have had little penetration for finding "nonshallow" bugs in most application domains.

8.1.2 Specifications from Documentation

Specifications might reasonably come from the library or interface writer. For example, one might expect the official API documentation to lay out all of the important specifications involving its member functions. In practice, this does not occur. As a case in point, the most recent version of the Java API documentation (which is currently the Java Platform Standard Edition 6) for `java.sql.Statement.close` states that it "releases this `Statement` object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed." This wording suggests that `close` is optional and that developers might harmlessly choose to wait for the resources to be reclaimed automatically. Conversely, the Oracle9*i* JDBC Developer's Guide and Reference makes the consequences of failing to do so clear: "The drivers do not have finalizer methods. ... If you do not explicitly close your `ResultSet` and `Statement` objects, serious memory leaks could occur. You could also run out of cursors in the database" [37]. Running out of database cursors lowers transactional throughput not just for the ill-behaved client but for all other clients sharing that database. Programmers are typically very concerned with closing these objects as quickly as possible, but nothing about that is mentioned in the official API documentation.

In addition, in the particular domain of security, many specifications forbid insecure practices that are officially allowed by the API or library. For example, there is nothing in the official IEEE Standard 1003.1 description of the `exec` family of system calls stating that `exec` should not be called before `setuid` root privileges have been dropped [38]. There are a few legitimate programs that fail to do so (for example, `su` and `sudo`). However, for most programs, failing to do so can lead to security flaws and the MOPS tool for finding security vulnerabilities checks that very policy [2]. More tellingly, despite the fact that `setuid` is a common system call with clear security implications that should be well documented, Chen et al. had to use a form of specification mining to find a formal model of `setuid` on Linux, Solaris, and FreeBSD systems [39]. Even security-critical functions in very popular interfaces are not adequately documented. Specification mining is thus quite important in practice.

Finally, in the rare cases when partial specifications are present in official documentation, they are not present in forms that can be automatically checked by bug-finding or software assurance tools. In order to be useful in this context, such natural language specifications would have to be converted to formal specifications. Unfortunately, previous research has shown that humans are poor at creating or debugging specifications [29]. Specification mining not only discovers important policies but also describes them formally for use by third-party tools. In practice, library specifications are not present, are conflicting, or are too informal, especially for security policies.

8.1.3 Lack of Traces

Despite these strong motivations for obtaining relevant specifications from programs, there has been only limited success in the area of specification mining. The primary reason for the inaccuracy of current specification mining is the lack of traces. Some traces can be gathered statically from the program source code or dynamically by running the program on indicative workloads. Both approaches present a skewed picture: Traces gathered from the program's source code overestimate unlikely or even infeasible paths, whereas traces gathered from workloads underrepresent exceptional situations that are critical to correctness. Both cases also present only a subset of all possible paths. In addition, the programs themselves typically violate the specifications. Traces are thus typically quite noisy and miners need many of them to work well.

Unfortunately, obtaining traces typically requires access to the source code or an instrumented binary, along with many indicative workloads. A fixed number of traces can be obtained from the source code and a given set of test cases. Given the size of typical test suites, that number is insufficient, in practice, for precise mining [24], [29].

Our solution is to combine the traces from multiple separate programs to learn policies about any components or interfaces that they share. For example, traces from two Java programs that both use the `JDBC` database interface could be combined to mine specifications about the proper use of that interface. Similarly, multiple programs written against the `Win32` API or the C standard library could be used to learn policies in those domains. Policies learned from the combined set of traces can be applied to both programs, finding more bugs in each.

8.1.4 Sharing Traces

Despite this incentive of working together, companies may be unwilling to share their source code, binaries, or test cases. We give both privacy and security reasons for why it is problematic to share source code and traces.

From a privacy perspective, a company may not release source code if it is possible that a subsequently discovered specification exposes that the company has particularly

buggy code. For example, if a specification is released which says that function A must precede function B and a company's source code has many examples of B preceding A, then it is possible to deduce how buggy the company's code is. Furthermore, it is particularly troubling if it is possible to deduce that company X has buggier code than company Y. Note that this information can even be deduced from the traces, that is, without the source code, since a trace includes the number of times that B precedes A.

From a security perspective, making the source code available can, if the company is not adequately prepared, make it easier for attackers to find and exploit vulnerabilities. In the case of SQL injection attacks, having the source code to a deployed application available would allow an attacker to tailor attacks to particular input handling routines that fail to sanitize the input correctly rather than having to exhaustively probe all input combinations. Similar information (for example, about control-flow and function call ordering) can be deduced from program traces: If an attacker can see that `inputA` and `inputB` are always followed by `sanitize` but that `inputC` never is, `inputC` can be targeted for attacks.

8.2 Secure Function Evaluation

Given the large benefit that sharing traces brings to specification mining, it is natural to imagine that many existing cryptographic techniques could be brought to bear to share traces without giving up privacy.

Current specification mining algorithms require that the sum of a collection of private values be computed. If there are n parties, each holding one of the private values $x_1, \ldots, x_n$, we explain how the sum can be securely computed without leaking anything beyond the sum.

The easiest solution is for party 1 to select a random value r and pass $x_1 + r$ to its neighbor party 2. The process repeats with party j passing $(\sum_{i=1}^{j} x_i) + r$ to its neighbor $j+1$. Eventually, party 1 receives from party n the value $(\sum_{i=1}^{n} x_i) + r$ and, by subtracting r from the value that it receives, learns and publishes the sum. This solution has the advantage of being simple and also not requiring any cryptographic assumptions. However, there is a single point of failure in that party 1 may learn the sum and not publish it. In addition, the technique is susceptible to collusion attacks: Parties i and $i+2$ can combine their values together to deduce party $(i+1)$'s private value.

Another solution is for each party i to select a random polynomial [40] p_i of degree n, where $p_i(0)$ is the private value x_i. Then, party i sends to each party j the value $p_i(j)$. Thus, party j can compute $q(j) = \sum_{k=1}^{n} p_k(j)$. Note that q is also a polynomial of degree n, but now $q(0) = \sum_i x_i$. Each party j then publishes $q(j)$ and, collectively, they have enough values to learn the coefficients of the polynomial q so that each party can compute $q(0)$. This solution does not require any cryptographic assumptions, but requires $O(n^2)$ communication overhead. In addition, $n-1$ parties may collude to determine one person's private value.

A final cryptographic solution uses homomorphic encryption techniques [40] that hinge on the hardness of computing discrete log. For simplicity, assume that computing the discrete log of a large number is difficult but that computing it for a small number is easy. Assume that each party i's private key is a_i, where $a = \sum_{i=1}^{n} a_i$ (a is not known to any party) and the public key is g^{a_i}, where g is a random integer selected from a sufficiently large range. Let r_i be a random value selected by party i. Party i publishes $(g^{r_i}, g^{ar_i} g^{x_i})$. All parties then compute the product of these values $(g^{\sum r_i}, g^{a \sum r_i} g^{\sum x_i})$. Each party then publishes $g^{a_i \sum r_i}$, and all parties compute the product $g^{a \sum r_i}$. Party i now divides $g^{a \sum r_i} g^{\sum x_i}$ by $g^{a \sum r_i}$ to obtain $g^{\sum x_i}$. Assuming that the x_i values are small, each party can now compute $\sum x_i$. Although this scheme is more efficient than the previous one, it requires synchronization among the parties and makes cryptographic assumptions.

We compare these secure computation solutions to our blurry trace solution. The secure computation assumption is that parties are committed to learning the exact sum. The privacy guarantee is stronger in that nothing is learned beyond the sum. However, these schemes are not resistant to collusion among $n-1$ parties. Since we only need an estimate of the sum, and only if there are enough parties, for our blurry trace scheme, even if $n-1$ parties collude, nothing can be learned from what a party publishes (by Lemma 2). Furthermore, the blurry trace solution does not hinge on any cryptographic assumptions.

9 Conclusion

We presented an algorithm for learning program specifications that preserves the privacy of the participant's traces. This allows mutually distrusting participants to reap almost all benefits of sharing basic trace data without allowing observers to draw unfavorable conclusions about the quality of their code. The algorithm allows participants to trade off utility in order to meet privacy requirements. Our experiments show that, even with strong privacy guarantees, it is possible to get useful results: We can preserve privacy while learning specifications that find 85 percent of the software bugs that could be found by forsaking privacy. The lack of specifications is currently a bottleneck in the use of automated bug-finding tools. Automated specification-mining techniques need many traces to be truly effective and companies are unwilling to give out their traces or their source code. We present a technique that gives participants the practical bug-finding and specification-mining benefits of sharing their data while maintaining their privacy.

Many avenues for future work remain. In terms of learning specifications, it would be interesting to design scalable algorithms that learn patterns beyond $(ab)^*$. In terms of privacy, although we have quantified an absolute worst-case loss of privacy when events are completely correlated (Lemma 3), it would be interesting if the analysis could be improved in the case when events are only somewhat correlated. Finally, our techniques currently assume independence among parties. However, it is possible for program traces from different parties to be correlated. A further investigation of the impact of correlation on privacy is warranted.

Acknowledgments

This research was supported in part by US National Science Foundation Grants CNS 0627523 and CNS 0716478 and US

Air Force Office of Scientific Research Grant BAA 06-028, as well as gifts from Microsoft Research. The information presented here does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred.

REFERENCES

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. Rajamani, and A. Ustuner, "Thorough Static Analysis of Device Drivers," *Proc. First European Systems Conf.*, pp. 103-122, Apr. 2006.

[2] H. Chen, D. Dean, and D. Wagner, "Model Checking One Million Lines of C Code," *Proc. 11th Ann. Network and Distributed System Security Symp.*, 2004.

[3] C. Conway, D. Dams, K. Namjoshi, and S. Edwards, "Incremental Algorithms for Interprocedural Analysis of Safety Properties," *Computer Aided Verification*, vol. 3576, pp. 449-461, 2005.

[4] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, F. Robby, and H. Zheng, "Bandera: Extracting Finite-State Models from Java Source Code," *Proc. 22nd Int'l Conf. Software Eng.*, pp. 762-765, 2000.

[5] M. Das, S. Lerner, and M. Seigle, "ESP: Path-Sensitive Program Verification in Polynomial Time," *SIGPLAN Notices*, vol. 37, no. 5, pp. 57-68, 2002.

[6] D. Engler, D. Chen, and A. Chou, "Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code," *Proc. 18th ACM Symp. Operating System Principles*, pp. 57-72, 2001.

[7] C. Flanagan, K. Rustan, M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata, "Extended Static Checking for Java," *Programming Language Design and Implementation*, pp. 234-245, 2002.

[8] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy Abstraction," *Principles of Programming Languages*, pp. 58-70, 2002.

[9] D. Hovemeyer and W. Pugh, "Finding Bugs Is Easy," *OOPSLA Companion*, pp. 132-136, 2004.

[10] W. Weimer and G. Necula, "Finding and Preventing Run-Time Error Handling Mistakes," *Object-Oriented Programming Systems, Languages, and Applications*, pp. 419-431, 2004.

[11] B. Liblit, A. Aiken, A. Zheng, and M. Jordan, "Bug Isolation via Remote Program Sampling," *Programming Language Design and Implementation*, pp. 141-154, 2003.

[12] J. Whaley, M. Martin, and M. Lam, "Automatic Extraction of Object-Oriented Component Interfaces," *Proc. Int'l Symp. Software Testing and Analysis*, pp. 218-228, 2002.

[13] J. Nimmer and M. Ernst, "Automatic Generation of Program Specifications," *Proc. Int'l Symp. Software Testing and Analysis*, pp. 232-242, 2002.

[14] M. Taghdiri, "Inferring Specifications to Detect Errors in Code," *Automated Software Eng.*, pp. 144-153, 2004.

[15] T. Andrews, S. Qadeer, S. Rajamani, J. Rehof, and Y. Xie, "Zing: Exploiting Program Structure for Model Checking Concurrent Software," *Concurrency Theory*, pp. 1-15, 2004.

[16] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding Data Lifetime via Whole System Simulation," *Proc. 13th Usenix Security Symp.*, pp. 321-336, 2004.

[17] U. Shankar, K. Talwar, J.S. Foster, and D. Wagner, "Detecting Format String Vulnerabilities with Type Qualifiers," *Proc. 10th Usenix Security Symp.*, pp. 201-220, Aug. 2001.

[18] M. Ringenburg and D. Grossman, "Preventing Format-String Attacks via Automatic and Efficient Dynamic Checking," *Proc. 12th ACM Conf. Computer and Comm. Security*, pp. 354-363, 2005.

[19] R. Johnson and D. Wagner, "Finding User/Kernel Pointer Bugs with Type Inference," *Proc. 13th Usenix Security Symp.*, pp. 119-134, 2004.

[20] V. Livshits and M. Lam, "Finding Security Errors in Java Programs with Static Analysis," *Proc. 14th Usenix Security Symp.*, pp. 271-286, Aug. 2005.

[21] C. Flanagan, K. Rustan, and C. Leino, "Houdini: An Annotation Assistant for ESC/Java," *Formal Methods for Increasing Software Productivity*, pp. 500-517, 2001.

[22] R. Alur, P. Cerny, P. Madhusudan, and W. Nam, "Synthesis of Interface Specifications for Java Classes," *Proc. 32nd Ann. ACM Symp. Principles of Programming Languages*, pp. 98-109, 2005.

[23] G. Ammons, R. Bodik, and J. Larus, "Mining Specifications," *Proc. 32nd Ann. ACM Symp. Principles of Programming Languages*, pp. 4-16, 2002.

[24] W. Weimer and G. Necula, "Mining Temporal Specifications for Error Detection," *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 3440, pp. 461-476, 2005.

[25] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: Mining Temporal API Rules from Imperfect Traces," *Proc. 28th Int'l Conf. Software Eng.*, pp. 282-291, 2006.

[26] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-Guided Abstraction Refinement," *Computer-Aided Verification*, vol. 1855, pp. 154-169, 2000.

[27] A. Raman and J. Patrick, "The Sk-Strings Method for Inferring PFSA," *Proc. 14th Int'l Conf. Machine Learning Workshop Automata Induction, Grammatical Inference, and Language Acquisition*, 1997.

[28] O. Kupferman and R. Lampert, "On the Construction of Finite Automata for Safety Properties," *Automated Technology for Verification and Analysis*, vol. 4218, pp. 110-124, 2006.

[29] G. Ammons, D. Mandein, R. Bodik, and J. Larus, "Debugging Temporal Specifications with Concept Analysis," *Programming Language Design and Implementation*, pp. 182-195, 2003.

[30] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating Systems Errors," *Proc. 18th ACM Symp. Operating Systems Principles*, pp. 73-88, 2001.

[31] C. Artho, A. Biere, and S. Honiden, "Enforcer: Efficient Failure Injection," *Proc. 14th Int'l Symp. Formal Methods*, vol. 4085, pp. 412-427, 2006.

[32] O. Goldreich, S. Micali, and A. Wigderson, "How to Play Any Mental Game or a Completeness Theorem for Protocols with Honest Majority," *Proc. 19th ACM Symp. Theory of Computing*, pp. 218-229, 1987.

[33] A.C. Yao, "How to Generate and Exchange Secrets," *Proc. 27th Ann. Symp. Foundations of Computer Science*, pp. 162-167, 1986.

[34] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, N. Mishra, R. Motwani, U. Srivastava, D. Thomas, J. Widom, and Y. Xu, "Enabling Privacy for the Paranoids," *Proc. 30th Int'l Conf. Very Large Databases*, vision paper, 2004.

[35] S. Warner, "Randomized Response: A Survey Technique for Eliminating Error Answer Bias," *J. Am. Statistical Assoc.*, 1965.

[36] N. Mishra and M. Sandler, "Privacy via Pseudorandom Sketches," *Proc. 25th ACM Symp. Principles of Database Systems*, pp. 143-152, 2006.

[37] E. Perry, M. Sanko, B. Wright, and T. Pfaeffle, "Oracle9i JDBC Developer's Guide and Reference," Technical Report A96654-01, http://www.oracle.com, Mar. 2002.

[38] IEEE, "The Open Group Base Specification Issue 6, IEEE Std. 1003.1,"technical report, 2004.

[39] H. Chen, D. Wagner, and D. Dean, "Setuid Demystified," *Proc. 11th Usenix Security Symp.*, pp. 171-190, 2002.

[40] A. Shamir, "How to Share a Secret," *Comm. ACM*, vol. 22, pp. 612-613, Nov. 1979.

Westley Weimer received the BA degree in computer science and mathematics from Cornell University and the MS and PhD degrees from the University of California, Berkeley. He is currently an assistant professor at the University of Virginia. His main research interests include static and dynamic analyses and program transformations to improve software quality.

Nina Mishra received the PhD degree in computer science from the University of Illinois at Urbana-Champaign. She is an associate professor in the Department of Computer Science at the University of Virginia. She previously held joint appointments as a senior research scientist at the Hewlett-Packard Laboratories and as an acting faculty member at Stanford University. She is currently on leave in the Search Laboratories at Microsoft Research. She was the program chair of the 20th International Conference on Machine Learning and has served on numerous data mining and machine learning program committees. She also serves on the editorial boards of *Machine Learning*, the *IEEE Transactions on Knowledge and Data Engineering*, the *IEEE Intelligent Systems*, and the *Journal of Privacy and Confidentiality*. Her research interests include data mining and machine learning algorithms and privacy.