Distributed API Protocol Mining

Deng Chen[1, a], Yanduo Zhang[1, b], Wei Wei[1, c], Rongcun Wang[2, d], Xiaolin Li[1, e], Shixun Wang[3, f], Rubing Huang[4, g]

[1] Hubei Provincial Key Laboratory of Intelligent Robot, Wuhan Institute of Technology, Wuhan, P.R. China

[2] School of Computer Science and Technology, China University of Mining and Technology, Xuzhou, P.R. China

[3] School of Computer and Information Engineering, Henan Normal University, Xinxiang, P.R. China

[4] School of Computer Science and Telecommunication Engineering, Jiangsu University, Zhenjiang, P.R. China

[a] dchen@wit.edu.cn

[b] zhangyanduo@hotmail.com

[c] weiwei@huawei-elec.com

[d] rcwang@hust.edu.cn

[e] 932233986@qq.com

[f] wsxun@hust.edu.cn

[g] rbhuang@mail.ujs.edu.cn

***Abstract*—Dynamic Protocol Mining (DPM) techniques are a promising approach to infer useful API protocols automatically. However, their results are biased to input test cases and the instrumentation overhead discounts their usability in industrial practice. In this paper, we propose a distributed dynamic protocol mining framework NSpecMiner. Our framework is based on a client-server architecture, where the client tracer gathers Program Execution Traces (PETs) and sends them to the server for mining. Mined protocols are saved on the server to provide various kinds of remote services, such as API protocol retrieval and program verification, etc. Compared with local miners, NSpecMiner has many advantages: 1) A large number of diverse PETs are likely to be collected from multiple clients, which is essential for mining accurate and complete API protocols. 2) Instrumentation overhead can be balanced among multiple clients. 3) Via integrating the client tracer into widely used software, we can mine API protocols transparently and automatically without any human effort. To evaluate our technique, we performed a comparison test with a local miner ISpecMiner and NSpecMiner. Preliminary results show that our approach is effective to mine useful API protocols as local miners. While our method is able to gather PETs concurrently from multiple clients and other merits of the distributed technology will further benefit DPM significantly.**

Keywords—program temporal specification; program dynamic analysis; distributed specification mining; API protocol mining

I. INTRODUCTION

API protocols specify temporal constraints regarding the order of calls of API methods. For example, calling peek() on java.util.Stack without a preceding push() gives an EmptyStackException, and calling next() on java.util.Iterator without checking whether there is a next element with hasNext() can result in a NoSuchElementException. API clients that violate such protocols do not obtain the desired behaviors and may even crash the program [1].

API protocols are beneficial for many tasks of software development, such as program documentation, understanding, testing, verification, etc. However, programmers are reluctant to write API protocols. Even when available, there is no guarantee of their consistence, completeness, and correctness. Dynamic Protocol Mining (DPM) [2]-[5] is a promising approach to infer useful API protocols automatically. It always works in a two-phase mode: 1) Program Execution Traces (PETs) are gathered from client programs leveraging instrumentation techniques. 2) API protocols are synthesized from PETs based on various kinds of sequential data mining techniques. Compared with Static Protocol Mining techniques (SPM) [6]-[13], DPM can achieve more accurate results based on runtime information. Additionally, it can be used more extensively, especially when source codes are unavailable. Furthermore, many tricky problems with SPM, such as infeasible paths, complicated data structures and pointer aliasing can be avoided. However, the following drawbacks limit its applications in industrial practice: 1) The effect of DPM largely depends upon input test cases. If an improper set of test cases is selected, DPM may neglect many program paths and cause partial and inaccurate API protocols. 2) In order to gather PETs, DPM is required to run client programs, which may be difficult to be automated in some cases. 3) The runtime overhead caused by instrumentation techniques may discount the practical usability of DPM. 4) Existing DPM tools (such as ADABU [14]) always work in a manner as follows. First, they collect PETs from client programs using a tracer and then store the traces into a trace file. Then, they take the trace file as input and synthesize API protocols. Each run of a client program will generate a trace file and corresponding protocols. Results of multiple runs cannot be merged. What is worse is that mined protocols are biased to input trace files. Additionally, if PETs gathered from a program is scarce, we may achieve partial and inaccurate API protocols. Even though mined protocols can be merged, to gather enough PETs for mining, we should manually run a large number of programs one by one, which will result in significant manpower overhead and is unacceptable in practice.

This paper aims to improve the DPM techniques and promote their applications in industrial practice. We present a DPM framework NSpecMiner, which is based on a client-server architecture. The client is a tracer, which collects PETs and sends them to the server for mining. The server receives PETs and synthesizes API protocols. After that, the mined protocols are stored on the server to provide various kinds of remote services, such as API protocol retrieval and program verification, etc. The functioning of our framework is illustrated in Figure 1.

(DOI reference number: 10.18293/SEKE2017-089)

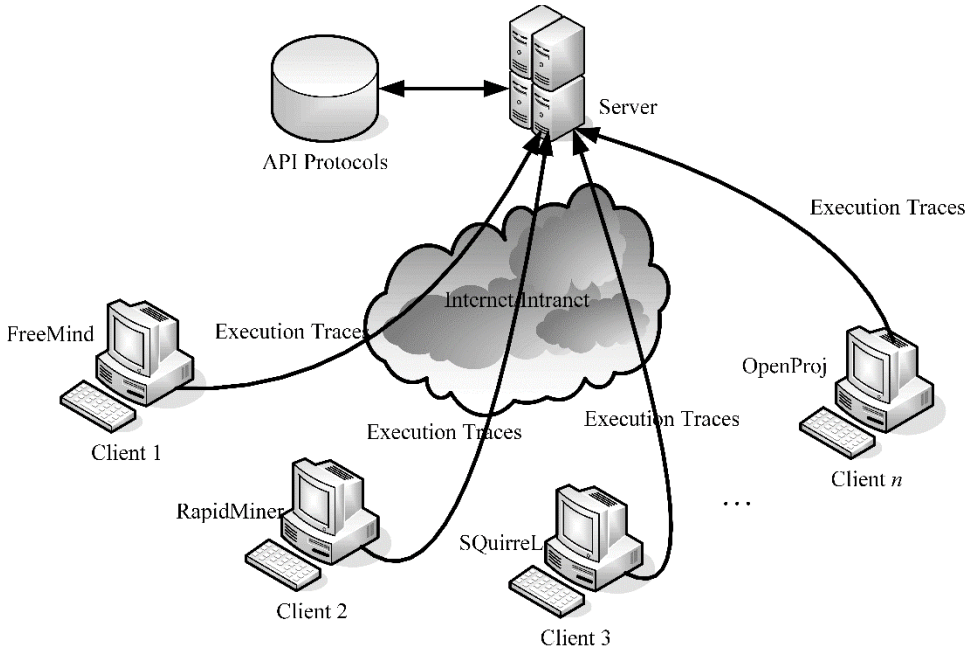

Figure 1 Functioning of NSpecMiner

Compared with local miners (such as ADABU), this architecture can benefit protocol mining in many ways: 1) Since the server can receive PETs from different clients and programs (as shown in Figure 1, the server receives PETs from *n* clients, which run different programs respectively, such as FreeMind, RapidMiner, SQuirreL and OpenProj), a large number of diverse PETs are likely to be collected efficiently, which is essential for mining accurate and complete API protocols. 2) The instrumentation overhead can be balanced among multiple clients based on a divide-and-conquer strategy. 3) This architecture is flexible, because client tracers based on whatever instrumentation techniques can send PETs to the server for mining if their formats satisfy requirements. 4) Via integrating the client tracer into daily used software, the task of mining protocols can be completely automated. On the client side, once the software is run by an end user, PETs will be sent to the server for mining automatically in the background. The whole mining process is transparent to software users. Little extra manpower is required to run client programs exclusively for gathering PETs. On the server side, thousands of millions of PETs may be received from different clients per second and perfect API protocols will be learned, which may cost several months by local miners.

The contributions of this paper are:

- A dynamic API protocol mining framework NSpecMiner based on a client-server architecture is proposed.
- A strategy of balancing instrumentation overhead is proposed.
- Experiments are conducted to evaluate our technique.

The rest of this paper is organized as follows: Section 2 presents the overview of our framework. Section 3 introduces an example API protocol mining technique used in this work. Section 4 elaborates our strategy of instrumentation balance. Section 5 presents our strategy of API protocol evolvement. Section 6 evaluates our technique and demonstrates preliminary results. Section 7 presents our conclusions and future work.

II. Overview of NSpecMiner

NSpecMiner is a generic dynamic API protocol mining framework. The distinguishing characteristic of it is that it is based on a client-server architecture, where clients collect PETs and send them to the server for mining. With the help of this framework, we can mine API protocols from application programs dynamically with little manpower overhead. Additionally, accurate and complete API protocols may be achieved.

The overview of NSpecMiner is illustrated in Figure 2, which mainly consists of a client and server. The client of NSpecMiner comprises a Proram Tracer and a Network Communication Module (NCM). The Program Tracer collects PETs from application programs and passes them to NCM. It gathers PETs based on instrumentation techniques, which insert binary codes into interested method bodies. Once an instrumented program is ran with test cases generated automatically or manually, the embedded codes will output information of method calls sequentially. Whatever instrumentation techniques and tools can be used in our framework (such as ASM [15], BCEL [16], Java agent [17] and Javassist [18]-[19]), provided that the format of output PETs can satisfy the requirement of NSpecMiner. After that, the client sends gathered PETs to the server via NCM, one method call after another.

The server receives method calls sequentially through NCM from multiple clients and passes them to the module of API Protocol Mining (APM). The APM is a key module for our

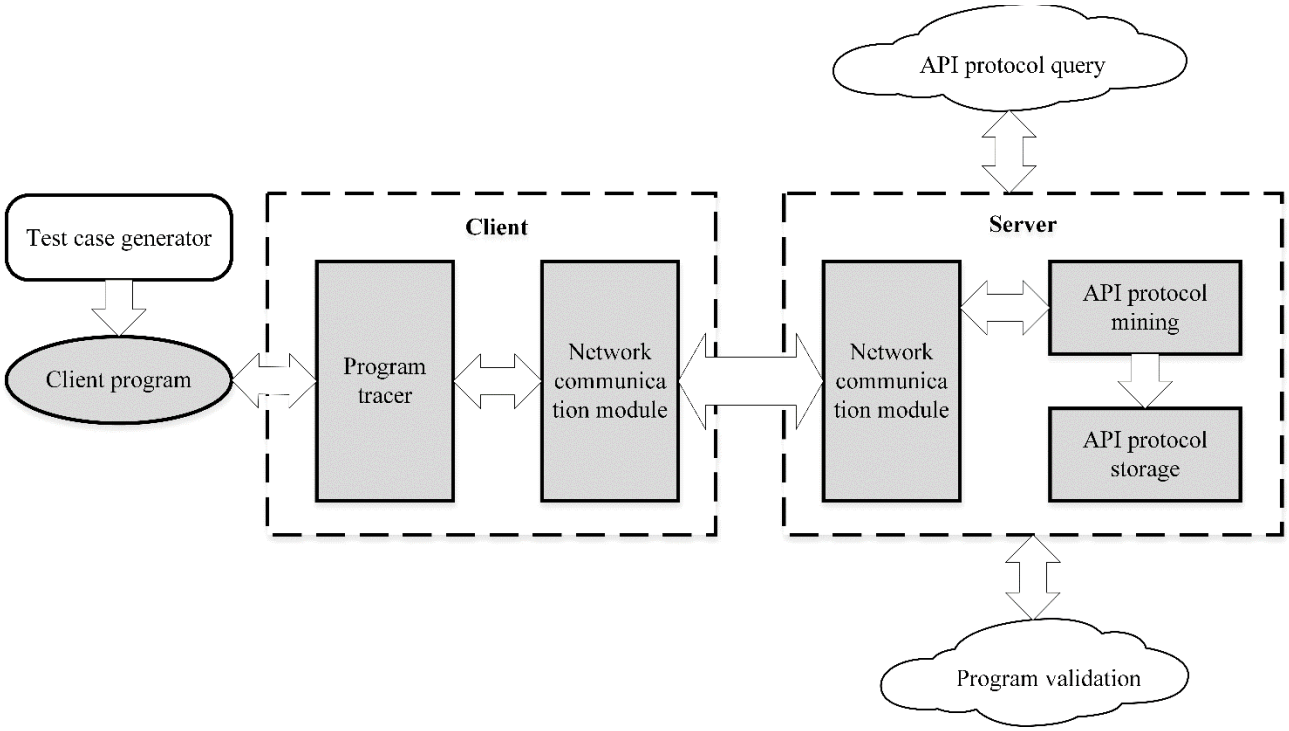

Figure 2 Overview of NSpecMiner

framework. It takes PETs as input and synthesizes API protocols based on sequential data mining techniques. For instance, [14], [20]-[23] mine API protocols based on Finite State Automaton (FSA). [24] models temporal specifications among Application Programming Interfaces (API) or Abstract Data Types (ADT) using Probabilistic Finite State Automaton (PFSA). Actually, whatever mining techniques based on sequential data can be used in our framework. In this paper, we utilize an online mining approach based on Markov model as an example to demonstrate the working principle of our framework. After that, NSpecMiner saves mined protocols on the sever through the module of API Protocol Storage (APS). Based on the protocols, our framework can provide various kinds of remote services, such as API protocol retrieval, program verification, etc.

NSpecMiner uses the TCP for communication and UDP is an inadvisable choice, because only the TCP can provide a reliable delivery service. Based on the TCP, all method calls sent sequentially to the network can be received in correct order on the server side, which is crucial for API protocol mining.

Via integrating the client tracer of NSpecMiner into widely used software, PETs will be sent to the server for mining automatically. The whole process is undergone in the background transparently. On the other hand, since the server can receive PETs from multiple clients (or programs), it is likely to achieve a large number of diverse PETs, which is essential for mining accurate and complete API protocols.

III. MINING API PROTOCOL

NSpecMiner is a generic framework, which can utilize various kinds of protocol mining techniques. In this paper, we use the online mining approach proposed by Chen et al. [25] as an example to demonstrate the working principle of our framework.

Chen et al. [25] mined API protocols based on an extended Markov model with final probability (MCF). The formal definition of MCF is given below:

Definition 1 (Markov chain with final probability). A *Markov Chain with Final Probability* (*MCF*) M is a 4-tuple (Q, τ, π, γ), where Q is a set of states, τ: $Q \times Q \rightarrow [0, 1]$ is the transition probability function, which is always described using a transition matrix P, π: $Q \rightarrow [0, 1]$ is the probability distribution over initial states, γ: $Q \rightarrow [0, 1]$ is the probability distribution over final states. The functions π and γ must satisfy the requirements: $\sum_{q \in Q} \pi(q) = 1$ and $\sum_{q \in Q} \gamma(q) = 1$.

Compared with traditional Markov model, MCF has an additional probability distribution over final states (Final Probability), which indicates how probable a chance process will end with a state.

Relying on the MCF, Chen et al. modeled API protocols by regarding states as methods and transitions as temporal relationships among methods. Figure 3 shows an example API protocol of Java class java.io.FileOutputStream described

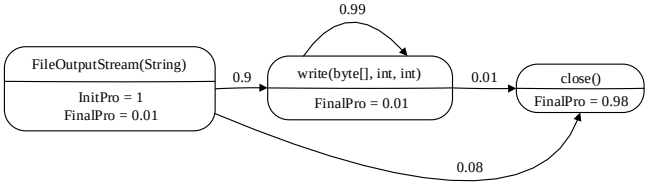

Figure 3 API protocol of Java class FileOutputStream described using MCF.

using the MCF. As we can see, the rounded rectangles are states, which are labeled with method signatures above a line. Arrows denote transitions, which are labeled with transition probabilities. InitPro and FinalPro are initial probability and final probability, respectively. What should be noted is that all states have properties InitPro and FinalPro. We omit the ones whose value is zero.

Given a repository of PETs *R*, Chen et al. learned MCFs from *R* using an online approach. In detail, they received a method call of each PET sequentially. For each method call, they evolved the corresponding MCF until it was good enough. The online approach has minimum space overhead. Additionally, since it can evolve API protocols persistently, accurate and complete protocols may be achieved.

Finally, they transformed a MCF to a Nondeterministic Finite-state Automaton (NFA) by discarding infrequent transitions and probabilities. The final API protocols described using NFA can be used for program verification, testing, documentation, etc.

IV. Instrumentation Balance

The time overhead incurred by instrumentation may cause performance issue to software running on the client machine and discount the practical usability of our framework. To mitigate the problem, we propose to balance instrumentation overhead among multiple clients.

Let's assume that we aim to mine API protocols of a set of classes *U*. It may cause much runtime overhead if we instrument all the classes in a single application program. To avoid this situation, we distribute the instrumentation task to multiple client programs via the instrumentation set. Let A be a set of application programs required to be instrumented. Given a program $a \in A$, the instrumentation set of a from U denoted by $\mathrm{IMTD}(a, U)$ is a subset of U, i.e., $\mathrm{IMTD}(a, U) \subseteq U$. The public methods of a class c will be instrumented in a run of program a, only if $c \in \mathrm{IMTD}(a, U)$. Obviously, via assigning program a a reasonable instrumentation set, we can confine its instrumentation overhead to an acceptable range. We perform this task based on the following method. First, we compute the size of the instrumentation set based on a function $\varphi: A \longrightarrow N$, which is given below:

$$\varphi(a) = \begin{cases} \varphi_1 & a \text{ is a hard real} - \text{time progam} \\ \varphi_2 & a \text{ is a soft real} - \text{time program} \\ \varphi_3 & a \text{ is a non} - \text{time} - \text{critical program} \end{cases}$$

where φ_1, φ_2 and φ_3 are specified constant values and satisfy the requirement $\varphi_1 < \varphi_2 < \varphi_3$. Our consideration is that less classes should be instrumented for time-critical programs than non-time-critical programs. After that, we select classes from *U* and add them to $\mathrm{IMTD}(a, U)$. What should be noted is that classes in *U* may not be covered by program a. To address this problem, we select classes from $\mathrm{CMTD}(a) \cap U$, where $\mathrm{CMTD}(a)$ is the set of classes covered by program a. Additionally, the following cases are considered:

1. $\mathrm{CMTD}(a) \cap U = \emptyset$, let $\mathrm{IMTD}(a, U) = \emptyset$.
2. $\mathrm{CMTD}(a) \cap U \neq \emptyset \wedge |\mathrm{CMTD}(a) \cap U| \leq \varphi(a)$, let $\mathrm{IMTD}(a, U) = \mathrm{CMTD}(a) \cap U$.
3. $\mathrm{CMTD}(a) \cap U \neq \emptyset \wedge |\mathrm{CMTD}(a) \cap U| > \varphi(a)$, we select $\varphi(a)$ classes from $\mathrm{CMTD}(a) \cap U$ based on a class selection algorithm and add them to $\mathrm{IMTD}(a, U)$.

In words, if the number of classes included in $\mathrm{CMTD}(a) \cap U$ is less than $\varphi(a)$, we add all classes in $\mathrm{CMTD}(a) \cap U$ to $\mathrm{IMTD}(a, U)$. Otherwise, we select as many as $\varphi(a)$ classes from $\mathrm{CMTD}(a) \cap U$.

In order to collect enough PETs, local protocol miners may instrument all classes in *U* at one time, which may cause much runtime overhead. Relying on the strategy of instrumentation balance, we can collect as many PETs as local protocol miners with little runtime overhead, which improves the practical usability of our framework significantly. It should be noted that our instrumentation balance strategy is based on the granularity of classes rather than methods. The reason for this is that partial instrumentation of a class will cause imperfect PETs which may lead to inaccurate API protocols [26].

V. API Protocol Evolvement

The network load overhead incurred by our framework may increase the cost of mining API protocols. In this section, we introduce our strategy of API protocol evolvement, which can reduce the network load overhead to some extent.

It should be noted that, there is no necessity to evolve API protocols persistently. Once a protocol is good enough, we can stop refining it, which can reduce the network load overhead and save computational resources on both client and server sides. In order to accomplish the task, a metric used to measure the goodness of API protocols is required. Currently, there does not exist such a metric and measuring the goodness of API protocols accurately and automatically is challenging. In this work, we perform the task approximately based on the following heuristic: let c be a class, p be the API protocol of c, which is described using a MCF: (Q, τ, π, γ). If the following requirements (*Good-Enough Requirement*) are satisfied, we believe p is good enough.

- $Q \supseteq \mathrm{PM}(c)$, where $\mathrm{PM}(c)$ denotes the set of public methods of class c (a single-object API protocol subsumes only public methods of a class);
- p keeps $\delta - constant$ in the continuous evolvement for T_e times, where T_e is the specified *Evolving Threshold*. Let $p_i: (Q_i, \tau_i, \pi_i, \gamma_i)$ and $p_{i+1}: (Q_{i+1}, \tau_{i+1}, \pi_{i+1}, \gamma_{i+1})$ be the API protocols achieved before and after the *i*th evolvement of p respectively. We say p is $\delta - constant$ in this evolvement if it satisfies the following requirements: 1) $Q_i = Q_{i+1}$; 2) $|\tau_i(t) - \tau_{i+1}(t)| \leq \delta$, where t denotes a common transition of p_i and p_{i+1}; 3) $|\pi_i(q) - \pi_{i+1}(q)| \leq \delta$, where q is a common state of p_i and p_{i+1}; and 4) $|\gamma_i(q) - \gamma_{i+1}(q)| \leq \delta$.

In detail, for each API protocol p, we maintain two variables n_p and n_e. The former records the number of methods included in p. The latter denotes the count of continuous evolvement, in which p is $\delta - constant$. At the end of each evolvement, we update n_p. As to n_e, we compare the protocols achieved before and after each evolvement. If p is $\delta - constant$, we have $n_e \leftarrow n_e + 1$, otherwise $n_e \leftarrow 0$. Once $n_p \geq n_c$ and $n_e \geq T_e$ where

n_c is the number of public methods included in class c, we notify all clients to stop instrumenting class c. Since no more PETs regarding c will be received, the evolvement of p on the server side will stop automatically.

In some cases, we need to restart the evolvement of good-enough protocols. For example, let's assume that p is a good-enough protocol of class c. The API protocol of c may be changed in the upgraded version of c. Thus, we should restart the evolvement of p until the good-enough requirement is satisfied in terms of the latest version of c. This task can be accomplished by notifying clients to restart instrumenting class c.

VI. Preliminary Results

In order to investigate the feasibility of our technique, we implemented NSpecMiner based on a previous prototype tool ISpecMiner [25] and called the novel tool ISpecMiner+. ISpecMiner and ISpecMiner+ are nearly the same: 1) both tools employ the Java agent [17] technique to instrument client programs; 2) both tools mine API protocols based on the online approach proposed by Chen et al. [25]. The only difference is that ISpecMiner is a local miner, while ISpecMiner+ is a distributed API protocol miner.

To evaluate our approach, we performed a comparison test with ISpecMiner and ISpecMiner+. Our experiment proceeds as follows. We used ISpecMiner and ISpecMiner+ to mine API protocols from a same set of programs. After that, we investigated API protocols achieved by ISpecMiner and ISpecMiner+, respectively. The experimental setup was exactly the same as that of [27]: 1) the subject programs used for mining were four real-world Java programs FreeMind, RapidMiner, SQuirreL SQL Client and OpenProj; and 2) 10 JDK classes from java.io and java.util were instrumented and investigated. Detailed information about the experimental setup please refer to [27]. ISpecMiner was configured to run each subject programs once sequentially. For each program, ISpecMiner instrumented all 10 JDK classes. Since ISpecMiner mines API protocols using an online approach, results of each ran will be merged automatically. ISpecMiner+ worked in a LAN environment similar to that shown in Figure 1, which consists of a server and four clients. Information of the server and client computers is summarized in Table I. On the server, ISpecMiner+ was configured to receive PETs from port 5123. Each client computer ran a subject program and instrumented all 10 JDK classes. To avoid biases, a subject program was feed with the same test cases under ISpecMiner and ISpecMiner+.

By analyzing experimental results, we found that API protocols mined by ISpecMiner and ISpecMiner+ were exactly the same. It indicates that NSpecMiner is effective to mine useful API protocols as local miners. While our technique is able to gather PETs concurrently from multiple clients. For each client user, only one subject program is required to run. If we integrate the client tracer of NSpecMiner into daily used software, API protocols can be synthesized on the server automatically and transparently. Little manpower overhead is required to exclusively run client programs for gathering PETs.

TABLE I. INFORMATION OF NETWORK COMPUTERS

Type	Number	Information
Server	1	Intel(R) Core(TM) i7-3770k CPU 3.9GHz, 8GB of memory, running Windows 7
Client	2	Intel(R) Core(TM) i3-2100 CPU 3.1GHz, 3GB of memory, running Windows XP
Client	1	Intel(R) Core(TM)2 Duo CPU 2.93GHz, 2GB of memory, running Windows XP
Client	1	Intel(R) Core(TM)2 Duo CPU 2.93GHz, 2GB of memory, running Windows 7

VII. Conclusions and Future Work

In this paper, we proposed a generic DPM framework NSpecMiner. The most distinguishing characteristic of our framework is that it is based on a client-server architecture. The distributed technology has been used widely in many aspects of software development, such as the Cooperative Bug Isolation Project [28], which collects program runtime information for tracking down bugs using a method similar to ours. However, to the best of our knowledge, NSpecMiner is the first DPM framework based on the distributed technology. As elaborated in this work, the client-server architecture can mitigate many limitations of DPM techniques, such as the overfitting problem incurred by scarce PETs, the difficulty to be automated, much runtime overhead caused by instrumentation, etc. Compared with local dynamic miners, NSpecMiner is able to mine accurate and complete API protocols with much lower cost, which increases the practical usability of DPM techniques significantly. In the evaluation, we performed a comparison test with local API protocol miners and our framework. Preliminary results show that our framework is effective to mine useful API protocols as local miners. While NSpecMiner is able to gather PETs concurrently from multiple clients and other advantages of the distributed technology will further benefit DPM significantly. If we integrate the client tracer of NSpecMiner into daily used software, API protocols can be synthesized on the server automatically and transparently. Little manpower overhead is required.

In conclusion, the distributed technology is significantly valuable for DPM and may promote its applications in industrial practice. Although the distributed technology is a common approach, there exist many particular challenges while using this approach for DPM, such as the strategy of instrumentation balance, the evolvement of API protocols, etc., which deserve much more research effort. Although some unforeseen issues regarding scalability, privacy and security may be raised when using our framework in practice, we are confident that they can be resolved relying on today's available technology. Additionally, along with the progress of network technology, our framework will be more useful. In this work, we presented a general view of NSpecMiner and only a preliminary evaluation result was given. Detailed discussions and more extensive evaluations of our framework are left as an extension of this work.

Acknowledgment

This work was supported by the Youths Science Foundation of Wuhan Institute of Technology (No. k201622), Surveying and Mapping Geographic Information Public Welfare Scientific Research Special Industry (No. 201412014), Educational Commission of Hubei Province (Q20151504), National Natural

Science Foundation of China (No. 41501505, 61502355 and 61502354) and the Key Program of Higher Education Institutions of Henan Province (No. 17A520040).

REFERENCES

[1] M. Pradel, T.R. Gross, "Leveraging test generation and specification mining for automated bug detection without false positives," in Proceedings of the 34th International Conference on Software Engineering, Zurich, Switzerland, pp. 288-298, 2012.

[2] M. P. Robillard, E. Bodden, D. Kawrykow, et al., "Automated API property inference techniques," IEEE Transactions on Software Engineering, 2013, 39 (5): 613-637.

[3] M. Gabel, Z. Su, "Javert: fully automatic mining of general temporal properties from dynamic traces," in Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, Atlanta, 2008.

[4] D. Chen, Y. Zhang, R. Wang, et al., "Extracting more object usage scenarios for API protocol mining," Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering, 2015: 607-612.

[5] D. Chen, Y. Zhang, R. Wang, et al., "Mining API protocols based on a balanced probabilistic model," Proceedings of the 12th International Conference on Fuzzy Systems and Knowledge Discovery, 2015: 2276 - 2282.

[6] M.K. Ramanathan, A. Grama, S. Jagannathan, "Static specification inference using predicate mining," SIGPLAN Not., vol. 42, pp. 123-134, 2007.

[7] S. Shoham, Y. Eran, S. Fink, et al., "Static specification mining using automata-based abstractions," in Proceedings of the 2007 International Symposium on Software Testing and Analysis, ACM, London, 2007.

[8] M. Di Penta, L. Cerulo, L. Aversano, "The life and death of statically detected vulnerabilities: An Empirical Study," Information and Software Technology, vol. 51, pp. 1469-1484, 2009.

[9] S. Thummalapenta, T. Xie, "Alattin: mining alternative patterns for defect detection," Automated Software Engineering, vol. 18, pp. 293-323, 2011.

[10] D. Lo, G. Ramalingam, V.P. Ranganath, et al., "Mining quantified temporal rules: formalism, algorithms, and evaluation," Science of Computer Programming, vol. 77, pp. 743-759, 2012.

[11] Z. Li, Y. Zhou, "PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code," SIGSOFT Softw. Eng. Notes, vol. 30, pp. 306-315, 2005.

[12] M.K. Ramanathan, A. Grama, S. Jagannathan, "Path-sensitive inference of function precedence protocols," in Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, 2007.

[13] M. Acharya, X. Tao, X. Jun, "Mining interface specifications for generating checkable robustness properties," in Proceedings of the 17th International Symposium on Software Reliability Engineering, 2006.

[14] V. Dallmeier, C. Lindig, A. Wasylkowski, et al., "Mining object behavior with ADABU," in Proceedings of the 2006 International Workshop on Dynamic Systems Analysis, ACM, Shanghai, 2006.

[15] ASM, http://asm.ow2.org, 2016.

[16] BCEL, http://commons.apache.org/proper/commons-bcel, 2016.

[17] P. Caserta, O. Zendra, "JBInsTrace: a tracer of Java and JRE classes at basic-block granularity by dynamically instrumenting bytecode," Science of Computer Programming, vol. 79, pp. 116-125, 2014.

[18] S. Chiba, M. Nishizawa, "An easy-to-use toolkit for efficient Java bytecode translators," in Proceedings of the 2nd International Conference on Generative Programming and Component Engineering," Springer-Verlag, New York, 2003.

[19] M. Tatsubori, T. Sasaki, S. Chiba, et al., "A bytecode translator for distributed execution of "legacy" Java software," in Proceedings of the 15th European Conference on Object-Oriented Programming, Springer-Verlag, 2001.

[20] A. Wasylkowski, "Mining object usage models," in Companion to the Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, 2007.

[21] D. Lorenzoli, L. Mariani, M. Pezz, "Automatic generation of software behavioral models," in Proceedings of the 30th International Conference on Software Engineering, ACM, Leipzig, 2008.

[22] R. Alur, P. Cerny, P. Madhusudan, et al., "Synthesis of interface specifications for Java classes," SIGPLAN Not., vol. 40, pp. 98-109, 2005.

[23] L.Mariani, F. Pastore, M. Pezze, "Dynamic Analysis for Diagnosing Integration Faults," IEEE Transactions on Software Engineering, 2011, 37(4): 486-508.

[24] G. Ammons, R. Bodik, J.R. Larus, "Mining specifications," SIGPLAN Not., vol. 37, pp. 4-16, 2002.

[25] D. Chen, R. Huang, B. Qu, et al., "Ming class temporal specification dynamically based on extended Markov model," International Journal of Software Engineering and Knowledge Engineering, 2015, 25(3): 573-604.

[26] J. Yang, D. Evans, D. Bhardwaj, et al., "Perracotta: mining temporal API rules from imperfect traces," in Proceedings of the 28th International Conference on Software Engineering, ACM, Shanghai, 2006.

[27] D. Chen, Y. Zhang, R. Wang, et al., "Mining universal specification based on probabilistic model," Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering, Pittsburgh, PA, USA, 2015: 471-476.

[28] The Cooperative Bug Isolation Project, http://research.cs.wisc.edu/cbi/, 2016.