SniP: An Efficient Stack Tracing Framework for Multi-threaded Programs

Arun KP
kparun@cse.iitk.ac.in
Indian Institute of Technology
Kanpur, India

Saurabh Kumar
skmtr@cse.iitk.ac.in
Indian Institute of Technology
Kanpur, India

Debadatta Mishra
deba@cse.iitk.ac.in
Indian Institute of Technology
Kanpur, India

Biswabandan Panda
biswa@cse.iitb.ac.in
Indian Institute of Technology
Bombay, India

ABSTRACT

Usage of the execution stack at run-time captures the dynamic state of programs and can be used to derive useful insights into the program behaviour. The stack usage information can be used to identify and debug performance and security aspects of applications. Binary run-time instrumentation techniques are well known to capture the memory access traces during program execution. Tracing the program in entirety and filtering out stack specific accesses is a commonly used technique for stack related analysis. However, applying vanilla tracing techniques (using tools like Intel Pin) for multi-threaded programs has challenges such as identifying the stack areas to perform efficient run-time tracing.

In this paper, we introduce `SniP`, an open-source stack tracing framework for multi-threaded programs built around Intel's binary instrumentation tool Pin. `SniP` provides a framework for efficient run-time tracing of stack areas used by multi-threaded applications by identifying the stack areas dynamically. The targeted tracing capability of SniP is demonstrated using a range of multi-threaded applications to show its efficacy in terms of trace size and time to trace. Compared to full program tracing using Pin, `SniP` achieves up to 75x reduction in terms of trace file size and up to 24x reduction in time to trace. `SniP` complements existing trace based stack usage analysis tools and we demonstrate that `SniP` can be easily integrated with the analysis framework through different use-cases.

CCS CONCEPTS

• **Computer systems organization → High-level language architectures**.

KEYWORDS

Multi-threaded programs, Run-time instrumentation, Stack tracing

ACM Reference Format:
Arun KP, Saurabh Kumar, Debadatta Mishra, and Biswabandan Panda. 2022. SniP: An Efficient Stack Tracing Framework for Multi-threaded Programs. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524842.3528499

1 INTRODUCTION

Stack is an important entity in software programs as it helps to efficiently implement language constructs such as subroutine call and return, allocation and freeing of local variables. Stack also plays a key role in program analysis as programs leave their footprint in the stack throughout their execution. Programmers can gain insights into the behaviour, security loopholes such as buffer overflow by analysing the stack usage. However, programmers face some unique challenges to perform analysis related to run-time stack usage. Unlike other program memory areas where the programmer explicitly control the usage (and can profile), the stack areas are hidden from the programmer. The usage of execution stack is transparent to programmers as the compiler inserts instructions to manage the stack for correct implementation of program logic (like function calls etc.).

These unique properties of stack makes it a fascinating element in the domain of program analysis. It is easy to observe that run-time stack usage can not be foreseen which makes static analysis techniques ineffective. Therefore, we depend on dynamic run-time techniques for stack analysis. Usage of debugging tools (e.g., GDB [20]) can provide a lot of insight into the stack usage but requires manual intervention and therefore, does not scale for long-running applications. On the other hand, run time memory access tracing techniques provide a lot of flexibility to perform automated tracing and analysis. Dynamic binary instrumentation (DBI) tools such as Intel Pin are widely used to trace the program at run time and perform offline analysis using variety of techniques [1, 8, 23].

DBI tools are very convenient as they require no preparation and can trace the entire program [12, 17]. One of the approaches adapted for run-time stack analysis is to perform full program tracing and filter stack specific accesses during the offline analysis phases. However, stack analysis through a trace-driven approach by tracing the program results in large trace files and higher tracing time (Section 3.1). In addition, the offline analysis process requires the stack virtual address ranges for filtering the trace and can

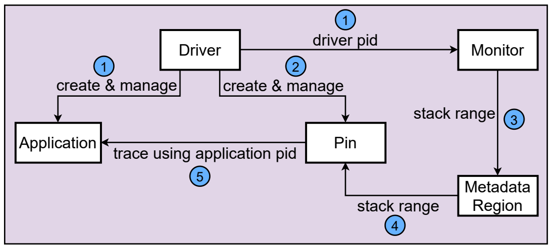


Figure 1: Schematic diagram of SniP

use operating system provided memory layout information for this purpose. However, capturing the stack range information for different threads in multi-threaded applications is challenging. The stack areas for threads can be allocated anywhere in the program address space depending on the state of the address space and OS support for dynamic allocation. For example, stack areas for the threads created using POSIX thread library in the Linux OS are allocated at run time in the non-contiguous areas in the address space.

In this paper, we take a different approach where we propose to filter the stack accesses at the time of tracing to avoid the additional overheads (trace size and tracing time) and remove the requirement of identifying the stack virtual address ranges during offline analysis. We propose techniques to identify and manage an information base for the stack address ranges of different threads which is consumed by the DBI tool to apply filters at the time of tracing. SniP, a **S**tack tracing framework for multi-threaded applications is built on top of Intel's Pin [12] (**niP**). SniP captures and manages the information regarding the stacks of different threads using an OS-level extension where the operations relating to the stack areas (starting from the creation) are monitored. Further, the up-to-date information related to the stack areas is shared with the user-space tracing process to enable target tracing.

To the best of our knowledge, SniP is the first framework for tracing stack in multi-threaded programs; showcasing reduced trace sizes (75x less trace file size with key-value store TinyDBM [21]) and tracing time (24x less tracing time with Python3 http server) compared to tracing the programs in entirety. Furthermore, with targeted tracing capability of SniP, the offline analysis process becomes simpler for multi-threaded applications which enables seamless integration with existing trace based analysis tools. SniP provides strength to the multi-threaded application stack analysis spectrum as multiple tools and use-cases can be built around SniP. We show two such sample use-cases using SniP (Section 3.2) to demonstrate its capabilities.

We release the tool for the community. The link to download the tool along with usage instructions is as follows, https://doi.org/10.5281/zenodo.6366894

2 DESIGN AND IMPLEMENTATION

The high-level design of SniP is shown in Figure 1. The major components of SniP are the user space program (Driver) and the OS module (Monitor). The user space program is responsible for executing the application and Intel's dynamic binary instrumentation tool Pin [12]. The monitor module captures application threads' stack range at the point of thread creation and store this information in the metadata region (Figure 1). This design enables the monitor module to record stack ranges of newly created threads throughout the application's lifetime. When tracing starts, Pin consumes the stack range information stored in the metadata region and generates trace only for stack accesses.

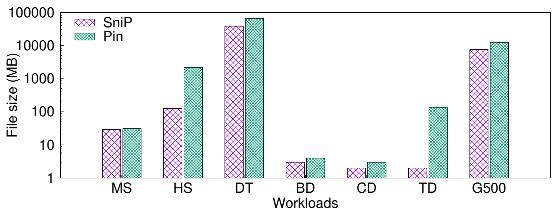
We implement SniP in the Linux OS (with kernel version 4.19.83). As we can see in Figure 1, the user space driver program creates two child processes, one for executing the application (to be traced) and the other for Pin. The driver program also passes its process ID (step ①) to monitor module (implemented as a kernel module) through a character device. This enables the monitor module to identify the application using the parent-child relationship between the driver and application process. The monitor module intercepts thread creation from the application process by hooking the `wake_up_new_task` function in `clone` system call handler of the Linux kernel and saves the stack range (step ③) of each application thread in the metadata region. The metadata region is exposed from the kernel module using the kernel `sysfs` API where the Pin process can access it using the file API. The driver passes the application process ID to Pin (step ②) for tracing. Pin starts reading stack ranges from metadata area (step ④) and generate output by tracing the application (step ⑤).

In the current implementation, we need to turn off Address Space Layout Randomization (ASLR) to prevent the `exec` system call (in create & manage) from changing the stack range of the application's main thread (after its forked from the driver process). We intend to address this limitation by hooking the `exec` system call handler in future. All the configurations for using the Pin tool remains unchanged in the proposed system and therefore, SniP does not hamper the vast feature set provided by Pin.

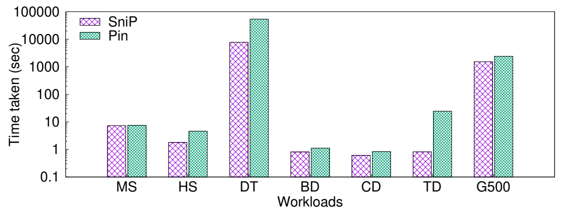
3 RESULTS

3.1 Stack tracing with SniP

To analyse the trace size and tracing time, we used the following workloads: merge-sort (MS) with four threads each sorting 250 numbers, Python3 default http-server (HS) used to download 4 files with each 200+ MB size, decision tree classifier (DT) from Python scikit-learn library used with 77280 training and 19315 testing samples, in-memory key-value stores BabyDBM (BD), CacheDBM (CD), TinyDBM (TD) from Tkrzw [21] performing 50 set and 50 get operations, Graph500 (G500) benchmark [15] BFS kernel run with scale parameter as 10. The benefits of using SniP for multi-threaded program stack tracing can be observed from the results presented in Figure 2 where we compare the trace file size and the tracing time taken with full program tracing using Pin and stack tracing using SniP. For long running applications such as python http-server, machine learning algorithms and in-memory key-value store applications, the difference between Pin and Snip is significant. For example, 17x and 2.5x reduction in the trace file size and the trace time is observed for http server workload. The benefit of SniP is marginal for short running applications with heavy stack usage, such as merge-sort which extensively use stack for recursive calls. To confirm the robustness, we also used SniP to trace long running applications such as MySQL with wikipedia [6] and memcached with YCSB workloads[5] (not shown here).

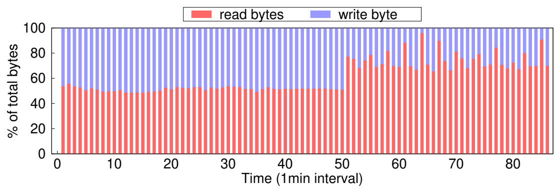


(a) Trace file size comparison (the lower the better)

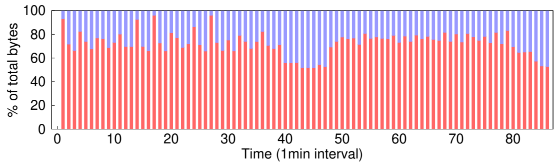


(b) Tracing time comparison (the lower the better)

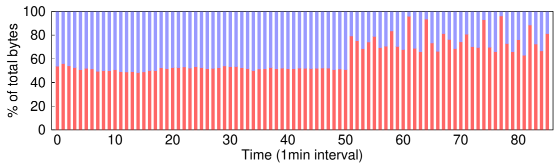
Figure 2: Comparison of size and time for SniP stack tracing w.r.t Pin full tracing [MS: Merge-Sort, HS: Python3 Http Server, DT: Decision Tree Classifier, BD: BabyDBM, CD: CacheDBM, TD: TinyDBM, G500: Graph500 BFS]



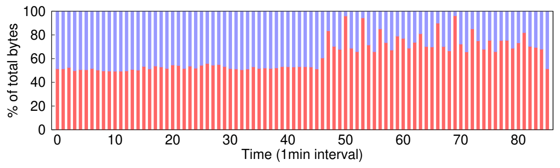
(a) Decision Tree Classifier



(b) Extra Trees Classifier



(c) Gradient Boosting Classifier



(d) Gaussian Naive Bayes

Figure 3: % of read and write access to stack of Machine Learning classification algorithms

3.2 Use cases

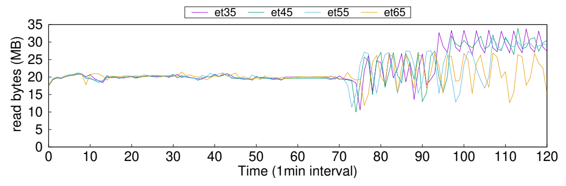
In this subsection, we show different usage scenarios of `SniP` for tracing the stack. We exhibit that `SniP` can be easily extended to build use-cases around stack analysis.

3.2.1 **Tracing ML Classification Algorithms:** The popularity of machine learning in a wide range of domains are driving software and hardware changes in computer science. ASICs and accelerators are designed to meet specific performance demands [18]. Understanding the memory access patterns of machine learning algorithms help designers to perform optimizations [14] at software and hardware levels. Moreover, understanding the memory access patterns is important for systems with non-volatile memory due to its internal micro-architecture difference with DRAM and asymmetric read, write access time [24]. First commercially available persistent memory, Intel's Optane DC persistent memory performance depends upon access size, access type (read vs write) and pattern [13, 25].

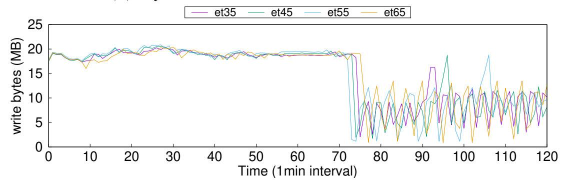
Given the popularity of machine learning algorithms and availability of non-volatile memory, it will be of great benefit to programmers to know the expected performance of machine learning algorithms executing on systems with non-volatile memory. In this context, we show a use-case for `SniP` by collecting the stack read, write access patterns of popular machine learning classification algorithms such as decision tree, extra trees, gradient boosting and gaussian naive bayes in Figure 3. These classification algorithms used 75 features from DeepDetect [11] and used 77280 samples for training and 19315 for testing. During this tracing, Decision Tree Classifier created 22, Extra Trees Classifier 14, Gradient Boosting Classifier 22 and Gaussian Naive Bayes 21 threads.

Figure 3 shows fraction of read and write operations to stack area in 1 minute intervals. We observed that the stack accesses in these algorithms are dominated by reads. Using `SniP`, we also performed a further study on the influence of feature set size on stack usage for ML classification algorithms by taking extra trees classifier as an example. Figure 4 shows that the feature set size did not influence stack read/write usage in the initial stage of the extra trees classifier algorithm, but impacted in the later stage of algorithm. Figure 4 also shows that there is an increase in bytes read from and decrease in bytes written to stack in the later stage of extra trees classification. We suspected it to be due to data access pattern difference in training and test phases of classification algorithm but confirmed that the pattern is present even after separating out training and test phases of extra trees classification which implies the access pattern is a property of the algorithm in this case.

3.2.2 **Detecting uninitialized memory usage in stack:** Memory corruption bugs such as buffer overflow, Use-After-Free (UAF) and uninitialized memory use happen due to incorrect programming practices or mistakes [9]. Static code analysis tools are effective for catching bugs based on a set of predefined rules but have a high false positive rate [2]. The uninitialized memory usage can also expose kernel data to user programs thus breaking the security guarantee [10]. The number of uninitialized memory bugs reported at CVE over the years in Figure 5 indicates the importance of detecting it.



(a) Bytes read from stack area in MB



(b) Bytes written into stack area in MB

Figure 4: Read and write behaviour of Extra Trees Classifier with varying feature set size [et35, et45, et55, et65 represents 35, 45, 55, 65 feature sizes]

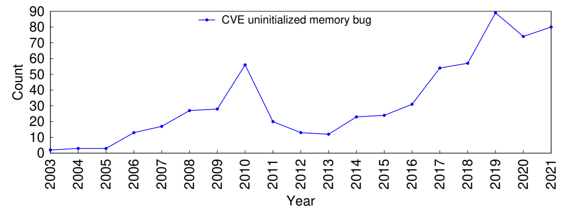


Figure 5: CVE uninitialized memory bugs reported over years

We show that SniP can be used to detect uninitialized memory bugs such as shown in listing 1.

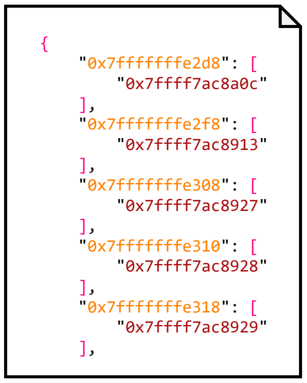
```c
int main(){
  int data[20];
  int i;
  for(i=0;i<20;i++)
    data[i] += 1;
  for(i=0;i<20;i++)
    printf("Value at data[%d] = %d\n",i,data[i]);
  return 0;
}
```

Listing 1: Prototype of uninitialized memory bug.

In this use-case, we developed a prototype to identify uninitialized memory bugs by parsing the stack trace generated by SniP. The parser tagged instances where read to any stack location happened before write, thus indicating uninitialized memory usage bug in the code. SniP's trace contained details such as access type (read/write), instruction address, memory access virtual address. The parser generated a JSON file containing uninitialized memory bug virtual address and instruction address as shown in Figure 6. Even though we parsed the SniP trace offline to detect bugs, this can be used to detect uninitialized memory bugs at run-time by running the parser alongside the traced program.

4 RELATED WORK

Static and dynamic analysis are two well known techniques for program analysis. Static analysis parses the code or uses abstract



```
{
    "0x7fffffffe2d8": [
        "0x7ffff7ac8a0c"
    ],
    "0x7fffffffe2f8": [
        "0x7ffff7ac8913"
    ],
    "0x7fffffffe308": [
        "0x7ffff7ac8927"
    ],
    "0x7fffffffe310": [
        "0x7ffff7ac8928"
    ],
    "0x7fffffffe318": [
        "0x7ffff7ac8929"
    ],
```

Figure 6: JSON file format, Python parser output of uninitialized memory bug

models whereas dynamic analysis executes the program and observes the run time behaviour, hence no abstractions or approximations are required [7]. Dynamic analysis requires inclusion of analysis routines within the program, which is called binary instrumentation. Binary instrumentation can be done statically (binary is modified before the program runs) or dynamically (modification occurs at run-time) [16].

Dynamic binary instrumentation (DBI) is very convenient for the users to trace and analyse programs as it requires no preparation and can be applied in a flexible manner. Valgrind [17] is a DBI framework that uses shadow values, which maintains a copy of the program state containing register values and user-mode address space. Dynamo performs run-time performance optimization. It generates an optimised version of the hot code sequence in the program to a code cache by interpreting the instructions [3]. Intel's Pin [12] uses dynamic compilation to instrument programs while they are executing. Users place analysis routines at point of interest in binary using instrumentation routines. Pin also allows attaching and detaching to a running process and we use this feature of Pin for tracing in SniP. Chabbi et. al. integrated call path collection library with Pin that collected call path context for each executed instruction using a shadow stack [4].

SniP generates trace for program stack accesses; stack trace holds important piece of information in debugging and program analysis as shown by Schroter et. al. [19] in their empirical study on the usefulness of stack trace. Experienced users can write their own tools for analysing trace generated by SniP or use existing tools such as STAT [1], which is used in debugging thousands of processes by sampling stack trace to form process equivalence class, then performing root cause analysis on the representative processes from equivalence class. Stack trace can also be used to compute similarities between bugs while reporting [22] or to identify dependency conflicts in projects using an automated approach [23]. Stack trace plays important role in system security as well, as highlighted by Feng et. al by using stack trace for anomaly detection, they extracted return address information from the stack for anomaly detection [8].

5 CONCLUSION

Stack holds important pieces of information to gain insights into the program behaviour which can help with debugging, security and performance analysis. In this paper, we discussed the challenges of tracing the stack accesses in multi-threaded applications using existing tools like Intel Pin. We introduced `SniP`, an efficient stack tracing framework for run-time tracing of application stack using techniques that combine tools like Pin and intelligent extensions to the OS. We implemented `SniP` in the Linux OS and demonstrated its efficacy in terms of its light tracing foot-print and flexibility in terms of applicability. Our experiments with a set of multi-threaded applications show that `SniP` not only outperforms Intel Pin in terms of resource usage but also makes the offline analysis of stack access traces comparatively simpler. Furthermore, we demonstrated the utility of `SniP` to perform stack analysis with contemporary application use cases by performing offline analysis of the stack access traces collected using `SniP`.

REFERENCES

[1] Dorian C Arnold, Dong H Ahn, Bronis R De Supinski, Gregory L Lee, Barton P Miller, and Martin Schulz. 2007. Stack trace analysis for large scale debugging. In *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–10.

[2] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. 2007. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 1–8.

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. 1–12.

[4] Milind Chabbi, Xu Liu, and John Mellor-Crummey. 2014. Call paths for pin tools. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 76–86.

[5] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[6] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288. http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[7] Michael D Ernst. 2003. Static and dynamic analysis: Synergy and duality.

[8] Henry Hanping Feng, Oleg M Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. 2003. Anomaly detection using call stack information. In *2003 Symposium on Security and Privacy, 2003.* IEEE, 62–75.

[9] Giovanni George, Jeremiah Kotey, Megan Ripley, Kazi Zakia Sultana, and Zadia Codabux. 2021. A Preliminary Study on Common Programming Mistakes that Lead to Buffer Overflow Vulnerability. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 1375–1380.

[10] Mateusz Jurczyk. 2018. Detecting kernel memory disclosure with x86 emulation and taint tracking.

[11] Saurabh Kumar, Debadatta Mishra, Biswabandan Panda, and Sandeep Kumar Shukla. 2021. DeepDetect: A Practical On-device Android Malware Detector. In *21st IEEE International Conference on Software Quality, Reliability and Security*. IEEE (in press).

[12] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[13] Tony Mason, Thaleia Dimitra Doudali, Margo Seltzer, and Ada Gavrilovska. 2020. Unexpected Performance of Intel® Optane™ DC Persistent Memory. *IEEE Computer Architecture Letters* 19, 1 (2020), 55–58. https://doi.org/10.1109/LCA.2020.2987303

[14] Sparsh Mittal, Poonam Rajput, and Sreenivas Subramoney. 2021. A survey of deep learning on CPUs: opportunities and co-optimizations. *IEEE Transactions on Neural Networks and Learning Systems* (2021).

[15] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. 2010. Introducing the graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.

[16] Nicholas Nethercote. 2004. *Dynamic binary analysis and instrumentation*. Technical Report. University of Cambridge, Computer Laboratory.

[17] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100.

[18] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. 2020. Survey of machine learning accelerators. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–12.

[19] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. 2010. Do stack traces help developers fix bugs?. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 118–121.

[20] Richard Stallman, Roland Pesch, Stan Shebs, et al. 1988. Debugging with GDB. *Free Software Foundation* 675 (1988).

[21] Tkzzw. 2021. Tkrzw: a set of implementations of DBM.

[22] Roman Vasiliev, Dmitrij Koznov, George Chernishev, Aleksandr Khvorov, Dmitry Luciv, and Nikita Povarov. 2020. TraceSim: a method for calculating stack trace similarity. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*. 25–30.

[23] Ying Wang, Ming Wen, Rongxin Wu, Zhenwei Liu, Shin Hwei Tan, Zhiliang Zhu, Hai Yu, and Shing-Chi Cheung. 2019. Could i have a stack trace to examine the dependency conflict issue?. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 572–583.

[24] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 496–508.

[25] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*. 169–182.