Statistical Reasoning About Programs

Marcel Böhme
Max Planck Institute for Security and Privacy, Germany
Monash University, Australia
marcel.boehme@acm.org

ABSTRACT

We discuss the advent of a new program analysis paradigm that allows anyone to make precise statements about the behavior of programs as they run in production across hundreds and millions of machines or devices. The scale-oblivious analysis leverages an almost inconceivable rate of user-generated program executions across large fleets to analyze programs of arbitrary size and composition with negligible performance overhead.

In this paper, we reflect on the program analysis problem, the prevalent paradigm, and the practical reality of program analysis at large software companies. We illustrate the new paradigm using several success stories and suggest a number of exciting new research directions.

ACM Reference Format:
Marcel Böhme. 2022. Statistical Reasoning About Programs. In *Proceedings of 44th International Conference on Software Engineering, NIER Track (Preprint (Accepted at ICSE'22-NIER))*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 FORMAL REASONING ABOUT PROGRAMS

Informally, *program analysis* aims to answer interesting questions about a program: Are there any bugs? Where are they located? What is the average execution performance? Where are the bottlenecks? Is there any information flow from a sensitive source to a public sink? Does this commit introduce any bugs? Do these pointers point to the same memory location? What is the type of this variable? Is this program statement reachable? What are the typical values of this variable? Can this assertion be violated?

Traditionally, we apply *formal reasoning* to analyze interesting properties of a program. A *program* consists of a set of instructions that tell the machine, which executes the program, precisely how to process a given input. The structure of a program is governed by syntactic rules of the programming language while the behavior of the program, i.e., its model of computation, is governed by the semantic rules of the language. To formally reason about a program means (i) to interpret its instructions according to the given semantic rules, (ii) to derive a model of computation that describes the relationship between the inputs and outputs of that program, and (iii) to compute the property of interest within this model of computation. We can reason about all executions (as in static analysis) or any specific execution (as in dynamic analysis).

There are several advantages to formal reasoning. We can analyze universal properties of a program that hold for all inputs. For instance, software verification aims to to prove the absence of bugs for all inputs, or to provide a counter-example. We can analyze the program irrespective of the machine that it is running on. In fact, we can analyze program properties at any level of abstraction. The program does not even need to be executable. Today, there are many industry-grade program analysis tools that leverage formal reasoning (specifically separation logic) to analyze program properties. For instance, the ErrorProne static analysis tool is routinely used at the scale of Google's two-billion-line codebase [28]. The Infer tool has substantial success at finding bugs at Facebook scale [5].

However, formal reasoning has fundamental limits. Landi argues in the "undecidability of static analysis" [20] that even a simple program analysis—such as determining whether two pointers point to the same memory location—is undecidable (may aliasing) or worse, uncomputable (must aliasing).[1] In fact, Rice's theorem implies that all interesting questions about the behavior of a program are undecidable. In practice, this has been addressed with a sacrifice in terms of soundness or completeness. For instance, the developers of Infer set the clear expectation that the tool may report many false alarms, does not handle certain language features, and can only report certain types of bugs.[2] The formally-minded reader might wonder about the formal guarantees of an analysis that trades soundness. These recent developments are a significant departure from the vision of formal reasoning for program analysis which Tony Hoare first laid out in 1969 [13]. In fact, Moshe Vardi recently reflected on Hoare's vison: "In retrospect, the hope for 'mathematical certainty' was idealized, and not fully realistic, I believe" [34].

Another challenge of formal reasoning is that the model of computation—which is extracted from the program—may be incomplete or incorrect. For instance, some third-party code may not be available or loaded dynamically, e.g., by reflection. Some program behavior is undefined whenever the semantic rules do not apply (e.g., when accessing arrays out of bounds) [35]. Peculiarities of the machine which executes the program are normally also missing from the extracted model of computation. This prevents us to formally reason about, e.g., micro-architectural attacks on the program's behavior [16, 19]. Moreover, large software systems today are *heterogeneous* and written in many languages while program analysis tools are often built to support the semantic rules of only a handful of programming languages [37]. So, how can we reduce this gap between the extracted model of computation and the actual execution of the program in production?

Preprint (Accepted at ICSE'22-NIER), May 2022, Pittsburgh, PA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] Ramalingam [27] provides an elegant proof of the undecidability of the aliasing problem by reduction to Post's correspondence problem.
[2] https://fbinfer.com/docs/limitations.html

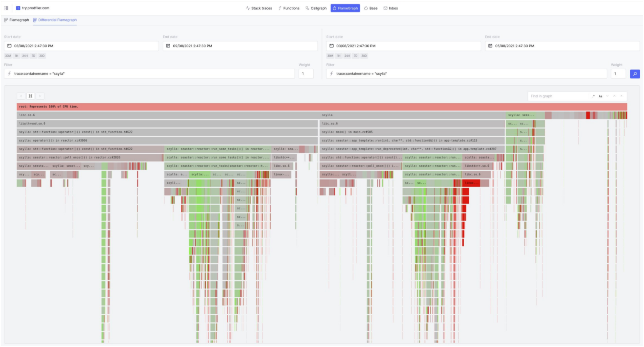

Figure 1: Differential flame graphs in Prodfiler, "the world's first whole-system multi-language continuous profiling platform [..] to unearth inefficiencies and optimization opportunities throughout your entire fleet" [9].

2 STATISTICAL REASONING BY SAMPLING-BASED PROGRAM ANALYSIS

Statistical reasoning about programs is enabled by a *scale-oblivious, sampling-based program analysis* approach. In the *observational setting*, the analysis measures the program property for a random sample of program executions. In the *experimentational setting*, the analysis iteratively generates and validates hypotheses about the program property by modifying and comparing forks of a random sample of executions. For instance, MutaFlow [23] detects information leakage by forking random executions, modifying information from private sources and monitoring information at public sinks across the original and forked execution.

At the ever-growing scale of industrial software systems, sampling-based program analysis can provide important insights of the program's runtime behavior in production that would be impossible to obtain by formal reasoning. Better efficiency can always be obtained by a lower sampling rate. However, unlike for analyses based on formal reasoning, *the (statistical) guarantees remain in tact* during the trade for efficiency.

Sound methodologies from statistics allow us to extrapolate, with quantifiable accuracy, from the properties of the observed executions to properties of the program as it is running in production. The probability to observe a given execution during production is called as *operational distribution*. In other words, using *samples* from the operational distribution, we can employ sound estimation methodologies to make claims about the program's behavior under the operational distribution itself. Going forward, to be clear, there are many statistical challenges to be tackled. Nevertheless, we believe that the statistical framework provides us with a new lens through which we can investigate the program analysis problem. Measuring the degree of uncertainty about certain facts is the backbone of any statistical analysis.

Why now? When Tony Hoare formulated his vision of formal reasoning about programs, only people in academic or research institutions had the opportunity for single-person use of a computer. Today, one *million* computers are sold *every day*.[3] The heterogeneity and scale of today's software systems imposes an ostensibly insurmountable challenge on program analysis. Yet, new technology also poses hitherto uncharted opportunities for program analysis:

- **Virtualization and Ultra-Large-Scale** [25, 33]. The need to scale a software system across an arbitrary number of machines has lead to the innovation of continuous and elastic deployment of software systems (e.g., to efficiently deploy a program analysis across all machines in a data center).
- **Compiler passes** [30]. Anyone can implement custom compiler passes that instrument any program when it is compiled.
- **Open-source OS kernel** [8]. Anyone can contribute to an open-source operating system kernel (e.g, to make no-overhead bug detection available to all programs running on the Linux OS).
- **Hardware-Assisted program analysis** [31]. Software companies are working with hardware manufacturers to enable hardware-assisted support for program analysis. This allows to make certain kinds of analysis very efficient.

In the following, we will discuss three isolated efforts that together point to a larger paradigm shift in program analysis for industrial-scale software systems.

2.1 Fleet-Wide Profiling in Production

Optimyze Cloud's ProdFiler [9] is a fleet-wide whole-system continuous profiling platform developed by Optimyze Cloud. Figure 1 shows an example of a differential flame graph generated by ProdFiler. A *flame graph* shows the executed stack traces for a program in the order of their execution. The y-axis shows the depth of the call stack (one function calls another) while the y-axis shows the order of the calls (one function is called after the other). In this *differential call graph*, the red color represents calls stacks that take longer to return while green call stacks return more quickly than in the reference profile (here, when the workload was smaller). Performance profiling is a simple, yet very powerful program analysis. Gprof [12] was one of the first performance profilers and it already used a sampling-based approach to measure the execution time of functions and basic blocks with low overhead. ProdFiler essentially scales the gprof ideas from a single program running on a single machine to many programs running on an entire fleet of machines.

ProdFiler monitors the execution of a programs using a Linux kernel extension called eBPF[4] which allows run sandboxed programs directly in the kernel. With this technology, ProdFiler can avoid the need for source code, instrumentation, debug symbols, or binary rewriting. Despite being always on even in production, the startup company reports extremely low overhead for their analyzes.

Google's GWP [33] goes beyond performance profiling and collects information such as stack traces, hardware events, lock contention profiles, heap profiles, and kernel events. GWP is routinely used at Google and was introduced as the first fleet-wide continuous profiling tool for cloud applications that are run in their data centers. In addition to performance bottlenecks, GWP can identify contended locks, micro-architectural peculiarities, the worst memory hogs, and the best memory allocation scheme for an application. When GWP was first published, it was run on thousands of machines across several data centers.

[3] https://www.tomshardware.com/news/over-1-million-pcs-sold-every-day

[4] https://ebpf.io/

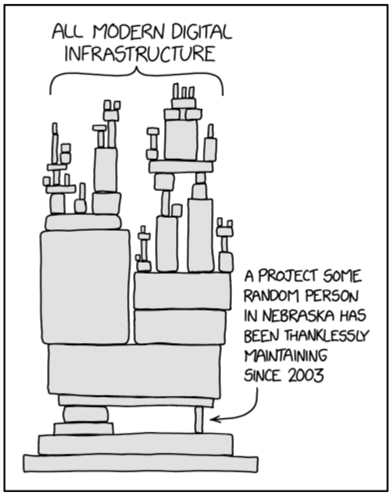

Figure 2: 'Dependency' by Randall Munroe (CC BY-NC 2.5). https://xkcd.com/2347/

2.2 Fleet-Wide Bug Detection in Production

Software security is important. At Google, fuzzing is the *first* line of defense.[5] A fuzzer generates inputs for a program while a bug detector crashes the program whenever a bug is found. For instance, ASAN [30] detects memory-safety issues that could otherwise be exploited to mount arbitrary code execution or privilege escalation attacks. The bug detector is instrumented directly into the program simply be enabling a compiler flag. However, not all bugs can be found during fuzzing and almost all software—that is running on the members of your fleet—remains untouched from your fuzzing efforts, but part of your supply chain (cf. Figure 2). Unfortunately, enabling bug detectors like ASAN in production is prohibitively expensive. In production, performance is critical. In this sense, sampling-based, no-overhead bug detection in production is the last line of defense.

GWP-ASAN [25, 32] is a sampling-based bug detection system that runs in *every* Chrome-browser, across *all* of Google's server-side applications, and *every* phone running Android 11 onwards [25, 32]. GWP-ASan uses an "electric fense" to guard some allocated memory. The allocated memory lives between two guard pages that throw a signal when accessed to detect buffer overflows. Freed memory is protected with `mprotect` to detect use-after-frees. A memory (de)allocation is subject to these measures with *a very low probability* (e.g., 1 in 10^5 executions). Hence, GWP-ASan can be employed in production with negligible performance overhead. However, across a large fleet of machines the signal is strong enough that bugs can reliably be discovered. GWP-ASan is on-by-default on every Chrome browser running on Windows and MacOS machines and on every phone running Android 11. In the past 18 months, it has found over 140 bugs in Chrome run in production, over 2k bugs in other Google products in production. Similar sampling-based bug detection techniques have recently been integrated into Firefox [14] and the Linux kernel [8].

[5] https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html

In terms of root-causing detected bugs, Liblit explored statistical approaches to localize a bug based on sparse (debugging) feedback generated in production from instrumentation trampolines [21, 22].

There are many opportunities of statistical reasoning for sampling-based bug detection. Using species richness estimation [3, 11], we could estimate the total number of bugs in the code base, given the sampling rate and bugs found over time. Using Good Turing theory [4], we could estimate the probability to discover a previously unseen bug. Using extreme value theory, we could estimate the likelihood of a production-disrupting or highly-critical type of bugs occurring. Using hypothesis testing, we could determine whether a bug has really been fixed. Applied statistics provides the right set of tools for us to answer such questions for systems of arbitrary scale, with arbitrarily high confidence.

2.3 Specification Mining for Microservice Architectures

Most industrial microservice architectures are complex networks of dependent microservices that are continuously updated and dynamically reorganized [37]. It is nearly impossible to untangle these dependencies, much less to analyze the entire system.

To tackle this challenge, the Akita analysis tool [38] builds powerful "API behavior models" from the message exchange among the constituent web services. These API models represent "endpoints, fields, data formats, latency, and more" [36]. To build these models, Akita passively watches the traffic on the network. The tool then draws a current map of the webservice-endpoints, the data that is sent, and the cross-service dependencies in the distributed software system. By tracking behavior models over time, Akita is able to detect breaking changes or data leaks before they cause any harm.

3 OPEN CHALLENGES AND OPPORTUNITIES

The ever-growing scale at which our software systems are run demand program analysis techniques that are *scale-oblivious*. On a single machine, a sampling-based program analysis can always trade a lower sampling rate for better efficiency. Across an entire fleet of machines, a sampling-based program analysis can employ statistical reasoning to quantify the (un)certainty with which we can make statements about interesting properties of the program as it is running in production. Unlike for formal-reasoning-based program analysis, the (statistical) guarantees remain in tact during the trade for better efficiency.

The only requirement for statistical reasoning is to inject analysis probes into the program or the machine executing the program. An *analysis probe* checks or measures certain properties of interest *during the execution* of the program in production. To extrapolate from the observed executions, applied statistics provides a rich toolset of methodologies that can be adopted for our purposes. To inject the required analysis probes, we can leverage recent technological advances. For instance, many compilers support custom instrumentation passes that can be used to inject analysis probes directly into the program binary during the build process [32]. Hypervisors, virtual machines, or operating system kernels can be modified to inject analysis probes into the execution of any program [8, 9, 33]. We can passively monitor the interaction of the program with other programs or the environment [37] to analyze properties of the program.

3.1 Open Opportunities

Foundations. For the research community there are opportunities to develop the probabilistic and statistical framework that will be underpinning a sound statistical reasoning about program properties. There is a large body of work in applied statistics [7], probability theory [1], and machine learning [24] that can be readily adopted to solve important program analysis questions that are currently out of reach from the formal perspective. For the statistician, the program analysis problem provides a diverse set of interesting statistical challenges that can be scientifically explored.

Conceptual Integration. We are also excited about the possibility to integrate formal and statistical reasoning about programs. From a formal perspective, statistical reasoning attaches probabilities to certain facts. For instance, a data flow that has not been observed in the sample has a certain probability to be observed in future executions, and this probability decreases as the sample increases. In statistics, this is the problem of estimating the missing probability mass, and many estimators have been proposed.

Techniques. The evolving need to analyze programs at the very-large-scale and the availability of emerging new technologies will generate a renewed research interest in the development of advanced sampling-based program analysis techniques (cf. Section 2). Given the tremendous success of GWP [33] and GWP-ASAN [25], we are excited about the opportunities of future scale-oblivious program analysis techniques.

Technical Integration. By integrating a sampling-based analysis into static analysis, we could overcome several well-known challenges of static analysis, like resolving the targets of register-indirect jumps, or making precise claims about program behavior in the presence of undefined behavior, dynamic loading, garbage collection, or reflection. Many static analyses techniques start by building a knowledge base that can then be efficiently queried. A sampling-based approach can feed facts into this knowledge base. The resulting analysis would be under-approximating (which is compatible with incorrectness-logic-based static analysis [26]).

3.2 Open Challenges

Heisenbugs. In addition to the opportunities, there are also exciting challenges that should be addressed by the research community. By moving the program analysis into production, we can derive claims about the program's behavior *in production.* However, techniques that are deployed in production should also minimize any unintended impact on the system in production. For instance, in the debugging research community, it is well known that some bugs cannot be observed *inside* the debugger but only outside. Unintentionally, the debugger modifies the behavior of the program. These bugs are called Heisenbugs. Program analysis techniques that are deployed in production should have a negligible impact on the program that is monitored.

Privacy concerns. Program analysis deployed in production should never reveal information about any specific user. If at all such data is recorded, any user data must be anonymized and aggregated to prevent privacy violations. Jin and Orso [15] explored an approach to reproduce (or "mimic") an interesting execution (here, a field-failure) on a local machine. In production, data is recorded at a reasonable degree of abstraction. They explored the effectiveness of their technique when recording only the point of failure, the call stack, the call sequence, or complete program trace, respectively.

Bounding the improbable and the adversarial. Of particular interest is the development of statistical methodologies to make claims about program properties that are benign but improbable, or that are outright adversarial. For instance, Serebryany clearly states that GWP-ASAN is not supposed to detect or prevent ongoing attacks [25]. The probability that GWP-ASan is effective for any given execution is just too small. On the other hand, if we do not observe an event despite a substantial sampling effort does not mean that the event is impossible. It might just be improbable. There is an entire field in applied statistics dedicated to assessing and quantifying the rare and extreme. For instance, rare event analysis [10] and extreme value theory [6] have become substantial fields of research with important applications, e.g., in economics, meteorology, actuarial science, physics, and ecology. In program analysis, extreme value theory has recently been introduced to tackle the problem of estimating a program's worst case execution time (WCET) [29].

4 DISCUSSION

We are excited about the opportunities and challenges of analyzing large-scale heterogeneous software systems in production. We argue that statistical reasoning is the only realistic approach to reason about programs in this setting. Statistical reasoning is enabled by a sampling-based analysis. For a better efficiency of the analysis, we can always trade a lower sampling rate. However, unlike for formal reasoning, which trades soundness or completeness for efficiency, statistical reasoning can always maintain the guarantees, if only at a quantifiable loss of accuracy.

We do not fully agree with Vardi's recent conclusion that "the hope for 'mathematical certainty' was idealized and not fully realistic" [34]. Formal reasoning has enabled the formal verification of an entire operating system microkernel [17, 18]. Recent advances particularly in separation logic are underpinning the tremendous success of static analysis tools like Facebook's Infer [5], Google's ErrorProne [28], or Github's CodeQL [2]. But we also believe that there is a growing practical need and emerging opportunities for a (statistically) sound program analysis that can be used for large, heterogeneous software systems, presently out of reach for static analysis. We are thrilled about the research opportunities that this emerging paradigm presents for the software engineering community and look forward to the development of the probabilistic and statistical foundations of program analysis.

ACKNOWLEDGMENTS

I would like to thank Peter O'Hearn of UCL and Lacework (and formerly of Facebook), Kostya Serebryany of Google, Lukas Dresel of UCSB, Carson Harmon of Square (formerly of Trail of Bits), and Toby Murray of University of Melbourne for their valuable feedback on earlier versions of this draft. This work grew out of my desire to recover some kind of guarantee for program analysis at scale, seeing that the most popular static analysis tools report bugs that do not exist or fail to report bugs that do indeed exist. Yet, we can't seem to quantify the degree to which our analyses are incorrect.

REFERENCES

[1] Dana Angluin. 1992. Computational Learning Theory: Survey and Selected Bibliography. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing (STOC '92)*. Association for Computing Machinery, New York, NY, USA, 351–369. https://doi.org/10.1145/129712.129746

[2] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2:1–2:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.2

[3] Marcel Böhme. 2018. STADS: Software Testing as Species Discovery. *ACM Transactions on Software Engineering and Methodology* 27, 2, Article 7 (June 2018), 52 pages. https://doi.org/10.1145/3210309

[4] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. 2021. Estimating Residual Risk in Greybox Fuzzing. In *Proceedings of the 15th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 494–504. https://doi.org/10.1145/3468264.3468570

[5] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6, Article 26 (Dec. 2011), 66 pages. https://doi.org/10.1145/2049697.2049700

[6] Enrique Castillo. 1988. *Extreme Value Theory in Engineering*. Academic Press, San Diego. https://doi.org/10.1016/B978-0-08-091725-2.50005-1

[7] Anne Chao and Colwell Robert K. 2017. Thirty years of progeny from Chao's inequality: Estimating and comparing richness with incidence data and incomplete sampling. *SORT-Statistics and Operations Research Transactions* 1, 1 (June 2017), 3–54.

[8] Linux Kernel Developers. 2021. Kernel Electric-Fence (KFENCE). https://www.kernel.org/doc/html/latest/dev-tools/kfence.html. [Online; accessed 15-Oct-2021].

[9] Thomas Dullien. 2021. Introducing Prodfiler. https://prodfiler.com/blog/introducing-prodfiler/. [Online; accessed 15-Oct-2021].

[10] M. Falk, J. Husler, J. Hüsler, and R.D. Reiss. 1994. *Laws of Small Numbers: Extremes and Rare Events*. Birkhäuser Verlag.

[11] N.J. Gotelli and R.K. Colwell. 2011. Estimating Species Richness. *Biological Diversity: Frontiers in Measurement and Assessment* (2011), 39–54.

[12] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (Boston, Massachusetts, USA) *(SIGPLAN '82)*. Association for Computing Machinery, New York, NY, USA, 120–126. https://doi.org/10.1145/800230.806987

[13] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

[14] Christian Holler. 2021. PHC (Probabilistic Heap Checker): a port of Chromium's GWP-ASan project to Firefox. https://bugzilla.mozilla.org/show_bug.cgi?id=1523268. [Online; accessed 15-Oct-2021].

[15] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing field failures for in-house debugging. In *Proceedings of the 34th International Conference on Software Engineering*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE, 474–484. https://doi.org/10.1109/ICSE.2012.6227168

[16] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors. *SIGARCH Comput. Archit. News* 42, 3 (June 2014), 361–372. https://doi.org/10.1145/2678373.2665726

[17] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive Formal Verification of an OS Microkernel. *ACM Trans. Comput. Syst.* 32, 1, Article 2 (Feb. 2014), 70 pages. https://doi.org/10.1145/2560537

[18] Gerwin Klein, June Andronick, Ihor Kuz, Toby Murray, Gernot Heiser, and Matthew Fernandez. 2018. Formally Verified Software in the Real World. *Commun. ACM* 61 (Oct. 2018), 68–77. Issue 10. https://doi.org/10.1145/3230627

[19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1–19. https://doi.org/10.1109/SP.2019.00002

[20] William Landi. 1992. Undecidability of Static Analysis. *ACM Letters on Programming Languages and Systems* 1, 4 (Dec. 1992), 323–337. https://doi.org/10.1145/161494.161501

[21] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. 2005. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 15–26. https://doi.org/10.1145/1065010.1065014

[22] Benjamin Robert Liblit. 2004. *Cooperative Bug Isolation*. Ph. D. Dissertation. University of California, Berkeley.

[23] Björn Mathis, Vitalii Avdiienko, Ezekiel O. Soremekun, Marcel Böhme, and Andreas Zeller. 2017. Detecting Information Flow by Mutating Input Data. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 263–273. https://doi.org/10.5555/3155562.3155598

[24] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. 2018. *Foundations of machine learning*. MIT press.

[25] Matt Morehouse, Mitch Phillips, and Kostya Serebryany. 2020. Crowdsourced bug detection in production: GWP-ASan and beyond. In *Proceedings of the C++ Russia*.

[26] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. 2020. Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 225–252.

[27] G. Ramalingam. 1994. The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.* 16, 5 (Sept. 1994), 1467–1471. https://doi.org/10.1145/186025.186041

[28] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (March 2018), 58–66. https://doi.org/10.1145/3188720

[29] Luca Santinelli, Jérôme Morio, Guillaume Dufour, and Damien Jacquemart. 2014. On the Sustainability of the Extreme Value Theory for WCET Estimation. In *14th International Workshop on Worst-Case Execution Time Analysis, WCET 2014, July 8, 2014, Ulm, Germany (OASICS, Vol. 39)*, Heiko Falk (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 21–30. https://doi.org/10.4230/OASIcs.WCET.2014.21

[30] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, USA, 28.

[31] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2019. Memory Tagging and how it improves C/C++ memory safety. *USENIX ;login* 44, 2 (2019). Issue Summer 2019.

[32] Vlad Tsyrklevich. 2021. GWP-ASan: Sampling heap memory error detection in-the-wild. https://www.chromium.org/Home/chromium-security/articles/gwp-asan. [Online; accessed 15-Oct-2021].

[33] E. Tune, G. Ren, T. Moseley, R. Hundt, Y. Shi, and S. Rus. 2010. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro* 30, 04 (jul 2010), 65–79. https://doi.org/10.1109/MM.2010.68

[34] Moshe Y. Vardi. 2021. Program Verification: Vision and Reality. *Commun. ACM* 64, 7 (June 2021), 5. https://doi.org/10.1145/3469113

[35] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2016. A Differential Approach to Undefined Behavior Detection. *Commun. ACM* 59, 3 (Feb. 2016), 99–106. https://doi.org/10.1145/2885256

[36] Jean Yang. 2021. Modeling API Traffic to Catch Breaking Changes. https://www.akitasoftware.com/blog-posts/modeling-api-traffic-to-catch-breaking-changes. [Online; accessed 15-Oct-2021].

[37] Jean Yang. 2021. The Software Heterogeneity Problem, or Why We Didn't Build on GraphQL. https://www.akitasoftware.com/blog-posts/the-software-heterogeneity-problem-or-why-we-didnt-build-on-graphql. [Online; accessed 15-Oct-2021].

[38] Jean Yang. 2021. Where My Specs At: From the Front to the Back. https://www.akitasoftware.com/blog-posts/where-my-specs-at-from-the-front-to-the-back. [Online; accessed 15-Oct-2021].