RESEARCH STATEMENT

AHMED TAMRAWI

Today's software systems are growing increasingly larger and more complex, and they are everywhere. For example, the recent version of the Linux kernel is about 20 MLOC. Software projects routinely measure in the millions of lines of code, span several programming languages, and are expected to run on a variety of platforms with various configurations. It is no great surprise that the mere mortals who develop these systems struggle to write code that is *reliable*, *secure*, and *efficient*. My research aims to answer the question: *how we can efficiently verify resilience of this mountain code to cyber-security attacks?*

My research makes it easier to build secure software systems. Society increasingly relies on software systems that handle sensitive information and safety-critical tasks: government, businesses, military, and private individuals all use software systems that manipulate confidential data and safety-critical tasks. Thus, the key challenge in building secure software systems is to develop software analysis techniques to: (1) define application-specific security guarantees; (2) express and understand the security requirements of applications; (3) develop interactive tools to provide human-comprehensible evidence to help analysts comprehend, visualize, and interact with detected anomalies, violations, and vulnerabilities for accurate auditing and verification.

My vision for the future is to broaden the scope and impact of program analysis. I want to make advanced program analysis techniques more useful and accessible for software developers to enable them build secure software systems, and to use my expertise in program analysis to push the boundaries of what can be discovered about program dynamics using automated techniques. In the remainder of this document, I will elaborate a selection of my research contributions and present some directions that I plan to pursue in the future.

Program Analysis for Software Security

With growing dependence on software in embedded and cyber-physical systems where vulnerabilities and malware can lead to disasters, efficient and accurate verification has become a crucial need for safety and cyber-security. Formal verification of large software has remained an elusive target, riddled with problems of low accuracy and high computational complexity [8, 6, 28, 29, 9]. The need for automating verification is undoubted [14, 13, 12], however human is indispensable to accurate real-world software verification. The automation should actually enable and simplify human crosschecking, which is especially important when the stakes are high. My research work in this area highlights a new frontier of software verification to ensure safety and security of critical software systems. The goal is about targeting automation to amplify human intelligence to scale it to large software. It projects the Intelligence Amplification (IA) vision propounded by Frederick Brooks [7].'

My research is based on the fact that software developers "*do not play dice*", and there is some hidden software model embedded in design documents that the developers embody into software. To this end, my research tries to leverage this knowledge by modeling software as graphs. Then, we develop techniques to abstract-out all irrelevant details in these graphs to result in compact and human-comprehensible graphs we call *evidence*. This evidence will be used to reason about behaviors of programs. We have developed algorithmic methods for answering queries about the static behavior of programs. Such queries have broad applications

in software engineering, including program verification (can a null pointer be dereferenced?), security analysis (can an adversary learn something about a customer's credit card information from the program's output?), and compiler optimization (is a given computation redundant?). Technology that is able to answer such questions quickly and accurately has the potential to revolutionize the way we build software.

The focus of my dissertation work was on evidence-enabled software verification [27]. It is about creating a powerful fusion of automation and human intelligence [16, 17] by incorporating algorithmic innovations to address the major challenges to advance the state of the art for accurate and scalable software verification where complete automation has remained intractable. The key is a mathematically rigorous notion of verification-critical evidence that the machine abstracts from software to empower human to reason with. The algorithmic innovation is to discover the patterns the developers have applied to manage complexity and leverage them. A pattern-based verification is crucial because the problem is intractable otherwise.

The papers [22, 11, 21, 20, 15] present a mathematical foundation to define relevant behaviors. Computing the relevant program behaviors involves: (a) computing the relevant program statements, (b) computing the relevant conditions to determine the feasibility of relevant behaviors, and (c) computing the relevant program behaviors. The papers introduce the Projected Control Graph (PCG) as an abstraction to directly compute the relevant behaviors for a fairly broad class of software safety and security problems. The paper presents an efficient algorithm to transform a Control Flow Graph (CFG) to PCG with complexity with $O(|V| + |E|)$, where $|V|$ and $|E|$ are respectively numbers of nodes and edges in the CFG. The PCG is human-comprehensible and much smaller than the corresponding CFG. I have also introduced an efficient PCG-based verification algorithm that leverages the PCG concept to define compact function summaries and used it to verify the lock/unlock pairing and allocation/deallocation pairing in the Linux kernel. The PCG-based verifier is able to verify 99.3% of the $66,609$ lock instances (from 3 different versions of the Linux kernel) in less than four hours. The PCG-based verifier was able to verify properties of (92.3%) of the allocation instances in one Linux version. This verification scalability, efficiency, and accuracy were previously out of the reach of state-of-the-art automated techniques such as BLAST [6]. The PCG-based verification resulted on reporting 7 instances of lock/unlock pairing violations and 50 instances of memory leaks to the Linux kernel community.

I have also worked on two DARPA projects APAC [1] and STAC [4] as part of my collaboration with Iowa State University team. In APAC project [1], the goal was to detect malware and security vulnerabilities in military android applications. Through the various live engagements at DARPA facility, we were presented with many military applications where sophisticated malware and security vulnerabilities were manually injected by third-party teams. In these engagements, we were asked to run our developed tools and analyses to detect these malware and security vulnerabilities. We were proud to be the leading team in this project in terms of efficiency and accuracy. The STAC project (ongoing) [4] is concerned about detecting Algorithmic Complexity (AC) and Side-Channel (SC) attacks in Java bytecode applications. The research goal was to build up on top of our analysis tools and techniques we used in APAC to detect: (1) hidden paths with higher algorithmic complexities that can be exercised by attackers, and (2) reveal clever adversary attacks that use traffic analysis via the study of the size and timing of network packets to reveal sensitive information. To this end, we developed an arsenal of program analysis, comprehension and visualization techniques to mitigate such attacks. This work is still in progress. Personally, I have engaged in two live engagements at DARPA facility where we were presented with curated apps with embedded AC and SC attacks. The goal is to use the developed techniques to detect embedded vulnerabilities.

Configuration Security and Build Code Analysis

Software building is the process that converts and integrates source code, libraries, and other data in a software project into stand-alone deliverables and executable files. The build process is managed by a build tool, i.e. a program that coordinates and controls others [3]. A build tool needs to execute the build commands according to the rules specified in build files, which are written in a build language supported by the tool. Popular build tools are make, ant, and maven. Prior research found that build maintenance could impose from $12\% - 36\%$ overhead on software development [18]. In a large-scale system, build files grow quickly and become very complex because they must support the building of the same software in multiple platforms with various configuration and environment parameters [10]. McIntosh *et al.* [19] found that from $4-27\%$ of tasks involving source code changes require an accompanied change in the related build code. They concluded that build code continually evolves and is likely to have defects due to high churn rate [19]. Importantly, those studies call for better tool support for build code.

My research focuses on developing techniques to support developers in dealing with complex build code. To this end, we have developed SYMake [23, 24], an infrastructure and tool for the analysis of build code in GNU make. SYMake includes Abstract Syntax Tree building module, a symbolic evaluation algorithm, and an evaluation trace building algorithm. We used SYMake to develop a tool to detect code smells and to support refactoring in Makefiles. Our evaluation on real-world Makefiles showed that our renaming tool is accurate and efficient, and that with SYMake, users could detect code smells and refactor Makefiles more accurately. Recently, we have submitted a proposal to the DARPA ConSec program [2]. The ConSec program aims at developing automated techniques to generate, deploy, and enforce configurations of components and subsystems for use in military platforms.

Software Maintenance

Software bugs are inevitable and bug fixing is an essential and costly phase during software development. Such defects are often reported in bug reports which are stored in an issue tracking system, or bug repository. Such reports need to be assigned to the most appropriate developers who will eventually fix the issue/bug reported. This process is often called Bug Triaging. Manual bug triaging is a difficult, expensive, and lengthy process, since it needs the bug triager to manually read, analyze, and assign bug fixers for each newly reported bug. Triagers can become overwhelmed by the number of reports added to the repository. Time and efforts spent into triaging typically diverts valuable resources away from the improvement of the product to the managing of the development process. To assist triagers and improve the bug triaging efficiency and reduce its cost, we focused on two aspects: (1) software tagging of bug reports and (2) automatic bug triaging.

Software tagging has been shown to be an efficient, lightweight social computing mechanism to improve different social and technical aspects of software development. Despite the importance of tags, there exists limited support for automatic tagging for software artifacts, especially during the evolutionary process of software development. We developed a novel, accurate, automatic tagging recommendation tool [5] that is able to take into account users feedbacks on tags, and is very efficient in coping with software evolution. The core technique is an automatic tagging algorithm that is based on fuzzy set theory. Our empirical evaluation on the real-world IBM Jazz project shows the usefulness and accuracy of our approach and tool. The tool we have developed is able to tag workitems based on previously manually tagged workitms. The goal was to allow developers to easily query workitem based on their social preferences when they tagged those workitems.

In terms of automatic bug triaging, we have developed Bugzie [26, 25], a novel approach for automatic bug triaging based on fuzzy set and cache-based modeling of the bug-fixing capability of developers. Our evaluation results on seven large-scale subject systems show that Bugzie achieves significantly higher levels of efficiency and correctness than existing state-of-the-art approaches.

References

[1] Automated Program Analysis for Cybersecurity (APAC). `https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-11-63/listing.html`, 2012.

[2] Configuration Security (ConSec). `https://www.darpa.mil/program/configuration-security`, 2012.

[3] Software Build. `https://en.wikipedia.org/wiki/Software_build`, 2012.

[4] Space/Time Analysis for Cybersecurity (STAC). `https://www.fbo.gov/spg/ODA/DARPA/CMO/DARPA-BAA-14-60/listing.html`, 2015.

[5] Jafar M Al-Kofahi, Ahmed Tamrawi, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Fuzzy set approach for automatic tagging in evolving software. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[6] Dirk Beyer and Alexander K. Petrenko. Linux driver verification. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, pages 1–6, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[7] Frederick P. Brooks, Jr. The computer scientist as toolsmith ii. *Commun. ACM*, 39(3):61–68, March 1996.

[8] C. Canal and A. Idani. *Software Engineering and Formal Methods: SEFM 2014 Collocated Workshops: HOFM, SAFOME, OpenCert, MoKMaSD, WS-FMDS, Grenoble, France, September 1-2, 2014, Revised Selected Papers*. Lecture Notes in Computer Science. Springer International Publishing, 2015.

[9] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *ACM SIGPLAN Notices*, volume 43, pages 270–280. ACM, 2008.

[10] Lorin Hochstein and Yang Jiao. The cost of the build tax in scientific software. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 384–387. IEEE, 2011.

[11] Benjamin Holland, Payas Awadhutkar, Suresh Kothari, Ahmed Tamrawi, and Jon Mathews. Comb: Computing relevant program behaviors. In *Companion Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018.

[12] Suraj Kothari, Ahmed Tamrawi, and Jon Mathews. Rethinking verification: Accuracy, efficiency, and scalability through human-machine collaboration. In *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pages 885–886. IEEE, 2016.

[13] Suraj Kothari, Ahmed Tamrawi, Jeremias Sauceda, and Jon Mathews. Let's verify linux: Accelerated learning of analytical reasoning through automation and collaboration. In *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pages 394–403. IEEE, 2016.

[14] Suresh Kothari, Payas Awadhutkar, and Ahmed Tamrawi. Insights for practicing engineers from a formal verification study of the linux kernel. In *Software Reliability Engineering Workshops (ISSREW), 2016 IEEE International Symposium on*, pages 264–270. IEEE, 2016.

[15] Suresh Kothari, Payas Awadhutkar, Ahmed Tamrawi, and Jon Mathews. Modeling Lessons from Verifying Large Software Systems for Safety and Security. In *Cyber-Physical Systems Special Track at Winter Simulation Conference (WSC)*, 2017. to appear.

[16] Suresh Kothari, Akshay Deepak, Ahmed Tamrawi, Benjamin Holland, and Sandeep Krishnan. A human-in-the-loop approach for resolving complex software anomalies. In *Systems, Man and Cybernetics (SMC), 2014 IEEE International Conference on*, pages 1971–1978. IEEE, 2014.

[17] Suresh Kothari, Ahmed Tamrawi, and Jon Mathews. Human-machine resolution of invisible control flow? In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–4. IEEE, 2016.

[18] Gary Kumfert and Tom Epperly. Software in the doe: The hidden overhead of"the build". Technical report, Lawrence Livermore National Lab., CA (US), 2002.

[19] Shane McIntosh, Bram Adams, Thanh HD Nguyen, Yasutaka Kamei, and Ahmed E Hassan. An empirical study of build maintenance effort. In *Proceedings of the 33rd international conference on software engineering*, pages 141–150. ACM, 2011.

[20] Ahmed Tamrawi and Suresh Kothari. Event-flow graphs for efficient path-sensitive analyses. *arXiv preprint arXiv:1404.1279*, 2014.

[21] Ahmed Tamrawi and Suresh Kothari. Projected control graph for accurate and efficient analysis of safety and security vulnerabilities. In *Software Engineering Conference (APSEC), 2016 23rd Asia-Pacific*, pages 113–120. IEEE, 2016.

[22] Ahmed Tamrawi and Suresh Kothari. Projected control graph for computing relevant program behaviors. *Science of Computer Programming*, 2018.

[23] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N Nguyen. Build code analysis with symbolic evaluation. In *Proceedings of the 34th International Conference on Software Engineering*, pages 650–660. IEEE Press, 2012.

[24] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N Nguyen. Symake: a build code analysis and refactoring tool for makefiles. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 366–369. ACM, 2012.

[25] Ahmed Tamrawi, Tung Thanh Nguyen, Jafar Al-Kofahi, and Tien N Nguyen. Fuzzy set-based automatic bug triaging (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 884–887. ACM, 2011.

[26] Ahmed Tamrawi, Tung Thanh Nguyen, Jafar M Al-Kofahi, and Tien N Nguyen. Fuzzy set and cache-based approach for bug triaging. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 365–375. ACM, 2011.

[27] Ahmed Yousef Tamrawi. Evidence-enabled verification for the linux kernel. 2016.

[28] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys (CSUR)*, 41(4):19, 2009.

[29] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(3):16, 2007.