RESEARCH STATEMENT

Ahmed Tamrawi

We are living in the information era and the only way to tame such an astronomical amount of information is through software. 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 surprise that developers face challenges in creating code that is *reliable*, *secure*, *efficient*, and *maintainable*. My overarching research goal is to address the critical question: *How can we efficiently verify the resilience of these extensive codebases against sophisticated cybersecurity attacks?*

My research aims to simplify the creation of secure software systems and assist developers in identifying security issues within their own systems. Society increasingly relies on software to handle sensitive information and safety-critical tasks across government, businesses, military, and private sectors. The key challenge in building and maintaining secure software systems is developing software analysis techniques that: (1) define application-specific security guarantees; (2) express and understand the security requirements of applications; and (3) develop interactive tools that provide human-comprehensible evidence to help analysts comprehend, visualize, and interact with detected anomalies, violations, vulnerabilities, and emergent unintended behaviors for seamless and accurate auditing and verification.

My vision for future research is to broaden the scope and impact of program analysis, pushing the boundaries of what can be discovered about program dynamics using novel machine-centric approaches. This involves making advanced program analysis techniques more useful and accessible for developers, enabling them to build secure software systems and efficiently audit and verify these systems against sophisticated cybersecurity attacks.

In addition to seeking an outstanding career at one of the world's leading universities, my overarching goal is to establish a robust research and development laboratory focused on creating novel and practical program analysis techniques to ensure the security of software systems and verify their resilience against sophisticated cyber-security threats. This new laboratory will aim to establish strong connections and collaborations with leading industry firms (such as Google, Microsoft, Amazon) and academia to define research goals and future directions.

The remainder of this document will highlight a selection of my research contributions and outline the future directions I plan to pursue.

Program Analysis for Software Security

With the growing dependence on software in embedded and cyber-physical systems, where vulnerabilities and malware can lead to disasters, efficient and accurate verification has become crucial for safety and cybersecurity [17]. Formal verification of large software systems has long been an elusive target, riddled by problems of low accuracy and high computational complexity [8, 6, 29, 30, 9]. The need for automating verification is undeniable [14, 13, 12]; however, human involvement remains indispensable for accurate real-world software verification [16]. Automation should enable and simplify human cross checking, especially when the stakes are high. My research work in this area highlights a new frontier of software analysis and verification aimed at ensuring the safety and security of critical software systems. The goal is to *leverage automation to amplify human intelligence*, scaling it to manage large software systems. This approach aligns with the Intelligence Amplification (IA) vision propounded by Frederick Brooks [7].

My research is grounded in the notion that software developers *do not play dice*; there is an inherent software model embedded in design documents that developers translate into software. My research focuses on leveraging this knowledge by modeling software as *graphs of graphs*. We then develop techniques to abstract out all irrelevant details from these graphs, resulting in compact and human-comprehensible graphs we call *evidence* [28, 12, 13, 23, 15]. This evidence is used to reason about the behaviors of programs. We have developed algorithmic methods for answering queries about the static behavior of programs. These queries have broad applications in software engineering, including program analysis and 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 can 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 [28]. This research aims to create a powerful fusion of automation and human intelligence [16, 18, 15] by incorporating algorithmic innovations to tackle major challenges and advance the state-of-the-art in 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, empowering humans to reason with it. The algorithmic innovation lies in discovering the patterns developers have applied to manage complexity and leveraging these patterns. Pattern-based verification is crucial because the problem is otherwise intractable.

The papers [23, 11, 22, 21, 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. These papers introduce the Projected Control Graph (PCG) as an abstraction to directly compute the relevant behaviors for a broad class of software safety and security problems. The papers present an efficient algorithm to transform a Control Flow Graph (CFG) into a PCG with complexity $O(|V| + |E|)$,

where |V| and |E| are the numbers of nodes and edges in the CFG, respectively. The PCG is human-comprehensible and significantly 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, using it to verify 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 three different versions of the Linux kernel) in less than four hours. Additionally, 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 beyond the reach of state-of-the-art automated techniques such as BLAST [6]. The PCG-based verification resulted in reporting seven instances of lock/unlock pairing violations and 50 instances of memory leaks to the Linux kernel community.

I have also worked on two high-profile DARPA projects, APAC [1] and STAC [4], as part of my collaboration with the Iowa State University team. In the APAC project [1], the goal was to detect malware and security vulnerabilities in military Android applications. Through various live engagements at DARPA facilities, we were presented with numerous military applications where sophisticated malware and security vulnerabilities were manually injected by third-party teams. During 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 [4] focuses on detecting Algorithmic Complexity (AC) and Side-Channel (SC) attacks in Java bytecode applications. The research goal was to build upon the analysis tools and techniques developed in APAC to: (1) detect hidden paths with higher algorithmic complexities that attackers could exploit, and (2) reveal clever adversary attacks using traffic analysis through the study of network packet size and timing to reveal sensitive information. To this end, we developed an arsenal of program analysis, comprehension, and visualization techniques to mitigate such attacks. Personally, I have participated in two live engagements at DARPA facilities, where we were presented with curated apps embedded with AC and SC attacks. The goal was to use the developed techniques to detect these embedded vulnerabilities.

Currently, I am leading the research and development effort at EnSoft Corp. to build a novel interactive analysis framework for binaries. The goal is to enable on-the-fly security vulnerabilities patching, making it easier for software practitioners to patch legacy systems and provide instant, small-sized security patches. This approach minimizes the risk of potential downtimes and reduces the effort required to reinstall systems from scratch.

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. This process is managed by a build tool, a program that coordinates and controls other tools [3]. A build tool executes the build commands according to the rules specified in build files, written in a build

language supported by the tool. Popular build tools include make, ant, and maven. Prior research has found that build maintenance can impose a 12% to 36% overhead on software development [19]. In large-scale systems, build files quickly grow in complexity as they must support building the same software across multiple platforms with various configuration and environment parameters [10]. McIntosh *et al.* [20] found that 4% to 27% of tasks involving source code changes also require changes in the related build code. They concluded that build code continually evolves and is likely to have defects due to a high churn rate [20]. These studies highlight the need for better tool support for build code.

My research focuses on developing techniques to support developers in managing complex build code. To this end, we have developed SYMake [24, 25], an infrastructure and tool for the analysis of build code in GNU make. SYMake includes an Abstract Syntax Tree building module, a symbolic evaluation algorithm, and an evaluation trace building algorithm. We used SYMake to develop a tool for detecting code smells and support refactoring in Makefiles. Our evaluation on real-world Makefiles showed that our renaming tool is accurate and efficient, and users could detect code smells and refactor Makefiles more accurately.

Software Maintenance

Software bugs are inevitable, and bug fixing is an essential and costly phase of software development. These defects are often reported in bug reports stored in an issue tracking system or bug repository. These reports need to be assigned to the most appropriate developers who will eventually fix the issues. This process, known as *bug triaging*, is challenging, expensive, and time-consuming, as it requires bug triagers to manually read, analyze, and assign bug fixers for each newly reported bug. Triagers can become overwhelmed by the volume of reports added to the repository. The time and effort spent on triaging typically divert valuable resources from product improvement to managing the development process. To assist triagers and enhance the efficiency and reduce the cost of bug triaging, we focused on two aspects: (1) software tagging of bug reports and (2) automatic bug triaging.

Software tagging has proven to be an efficient, lightweight social computing mechanism for improving various social and technical aspects of software development. Despite the importance of tags, there is limited support for automatic tagging of software artifacts, especially during the evolutionary process of software development. We developed a novel, accurate, automatic tagging recommendation tool [5] that considers user feedback on tags and efficiently copes with software evolution. The core technique is an automatic tagging algorithm based on fuzzy set theory. Our empirical evaluation on the real-world IBM Jazz project demonstrates the usefulness and accuracy of our approach and tool. The tool can tag work items based on previously manually tagged items, allowing developers to easily query work items based on their social preferences when they tagged those items.

Regarding automatic bug triaging, we developed Bugzie [27, 26], a novel approach for automatic bug triaging based on fuzzy set and cache-based modeling of developers' bug-fixing capabilities. 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.

Future Work

State-of-the-art approaches to software security often rely on black-box testing techniques, which typically overlook the internal workings of vulnerable software systems. These approaches mitigate security concerns by preventing suspicious access to the software without providing insights into the underlying vulnerabilities. Consequently, these systems remain susceptible to undetected and subtle potential attacks.

In our pursuit of enhancing software security, we aim to extend current methodologies by developing innovative analysis techniques and tools that leverage mathematical abstractions and machine-centric approaches. Our goal is to bridge the detection gap for sophisticated vulnerabilities, empowering software practitioners to audit and verify their systems with precision before releasing their code to customers. This requires integrating a hacker's mental model and sophisticated abstractions into our analysis framework, thereby amplifying the analyst's ability to detect intricate vulnerabilities.

Reflecting on my experiences with DARPA projects like APAC and STAC, and my tenure at EnSoft Corp., I have witnessed firsthand the critical need for robust security measures in software development. These experiences have fueled my passion for creating advanced security solutions that not only detect but also preempt vulnerabilities.

Moreover, securing build code presents a unique set of challenges due to the exponential combination of configurations and the variety of programming languages used within a single build system. To address these challenges, we aim to develop novel variability-aware analysis techniques that operate on graphs of heterogeneous build code artifacts. These artifacts include elements from different programming languages used in build systems, as well as configuration information that distinguishes various deployment environments.

In line with my vision of bridging academic research and industry practices, I propose the creation of a dedicated research lab focused on software security and verification. This lab would collaborate with leading industry firms and academic institutions to define cutting-edge research goals and develop practical solutions that ensure the security and resilience of software systems against sophisticated cyber-attacks. By fostering a collaborative environment, we can accelerate the advancement of software security technologies and create a safer digital future.

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] Al-Kofahi, Jafar M., Ahmed Tamrawi, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. "Fuzzy set approach for automatic tagging in evolving software." In 2010 IEEE international conference on software maintenance, pp. 1-10. IEEE, 2010.

[6] Margaria, Tiziana, and Bernhard Steffen. Leveraging applications of formal methods, verification and validation. Springer, 2008.

[7] Brooks Jr, Frederick P. "The computer scientist as toolsmith II." Communications of the ACM 39, no. 3 (1996): 61-68.

[8] Roggenbach, Markus, Antonio Cerone, Bernd-Holger Schlingloff, Gerardo Schneider, and Siraj Ahmed Shaikh. Formal Methods for Software Engineering. Springer, Switzerland, 2021.

[9]Dillig, Isil, Thomas Dillig, and Alex Aiken. "Sound, complete and scalable path-sensitive analysis." In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 270-280. 2008.

[10] Hochstein, Lorin, and Yang Jiao. "The cost of the build tax in scientific software." In 2011 International Symposium on Empirical Software Engineering and Measurement, pp. 384-387. IEEE, 2011.

[11] Holland, Benjamin, Payas Awadhutkar, Suresh Kothari, Ahmed Tamrawi, and Jon Mathews. "Comb: Computing relevant program behaviors." In Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, pp. 109-112. 2018.

[12] Kothari, Suresh, Ahmed Tamrawi, and Jon Mathews. "Rethinking verification: accuracy, efficiency and scalability through human-machine collaboration." In Proceedings of the 38th International Conference on Software Engineering Companion, pp. 885-886. 2016.

[13] Kothari, Suresh, Ahmed Tamrawi, Jeremias Sauceda, and Jon Mathews. "Let's verify linux: Accelerated learning of analytical reasoning through automation and collaboration." In Proceedings of the 38th International Conference on Software Engineering Companion, pp. 394-403. 2016.

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

[15] Kothari, Suresh, Payas Awadhutkar, Ahmed Tamrawi, and Jon Mathews. "Modeling lessons from verifying large software systems for safety and security." In 2017 Winter Simulation Conference (WSC), pp. 1431-1442. IEEE, 2017.

[16] Kothari, Suresh, Akshay Deepak, Ahmed Tamrawi, Benjamin Holland, and Sandeep Krishnan. "A “Human-in-the-loop” approach for resolving complex software anomalies." In 2014 IEEE international conference on systems, man, and cybernetics (SMC), pp. 1971-1978. IEEE, 2014.

[17] Kothari, Suresh, Ganesh Ram Santhanam, Payas Awadhutkar, Benjamin Holland, Jon Mathews, and Ahmed Tamrawi. "Catastrophic cyber-physical malware." Versatile Cybersecurity (2018): 201-255.

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

[19] Kumfert, Gary, and Tom Epperly. Software in the DOE: The Hidden Overhead of"The Build". No. UCRL-ID-147343. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2002.

[20] McIntosh, Shane, 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, pp. 141-150. 2011.

[21] Tamrawi, Ahmed, and Suresh Kothari. "Event-flow graphs for efficient path-sensitive analyses." arXiv preprint arXiv:1404.1279 (2014).

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

[23] Tamrawi, Ahmed, and Suresh Kothari. "Projected control graph for computing relevant program behaviors." Science of Computer Programming 163 (2018): 93-114.

[24] Tamrawi, Ahmed, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N. Nguyen. "Build code analysis with symbolic evaluation." In 2012 34th International Conference on Software Engineering (ICSE), pp. 650-660. IEEE, 2012.

[25] Tamrawi, Ahmed, 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, pp. 366-369. 2012.

[26] Tamrawi, Ahmed, 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, pp. 884-887. 2011.

[27] Tamrawi, Ahmed, 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, pp. 365-375. 2011.

[28] Tamrawi, Ahmed. "Evidence-enabled verification for the Linux kernel." (2016).

[29] Woodcock, Jim, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. "Formal methods: Practice and experience." ACM computing surveys (CSUR) 41, no. 4 (2009): 1-36.

[30] Xie, Yichen, and Alex Aiken. "Saturn: A scalable framework for error detection using boolean satisfiability." ACM Transactions on Programming Languages and Systems (TOPLAS) 29, no. 3 (2007): 16-es.