Maverick Woo 

Research Statement

With the ever-increasing computerization in our society, the security of our software has become one of the most important aspects of our daily lives. Unfortunately, the software currently running on a large number of legacy systems was designed for outdated threat models, and even newly-developed software may be driven by economic factors to favor new features over security. As such, software vulnerabilities have become an everyday reality and the need to uncover them before attackers do has become increasingly critical to our safety and prosperity.

My research focuses on designing new algorithms and systems to improve software security processes, such as vulnerability discovery, software testing, and binary analysis. I generally favor approaches with solid theoretical underpinnings that can predict and explain why they should outperform prior work; however, I am also a pragmatist at heart and I embrace practical and cost-effective heuristics when they are backed by rigorous validations using real-world data. In the following, I will first focus on two lines of work that I have been conducting with my collaborators over the last few years. Then I will briefly discuss my other research projects and conclude with my future research directions.

**Fuzzing.** Intuitively, fuzzing refers to the process of repeatedly running a target program with automatically generated and potentially malformed inputs. The goal is to trigger observable bugs such as crashes. Fuzzing is highly popular among penetration testers for vulnerability discovery. One reason is because fuzzing is relatively easy to deploy. At its most basic, fuzzing can be performed with nothing other than the ability to generate random inputs and monitor the target program for crashes. A slightly more sophisticated variant known as “Black-box Mutational Fuzzing” starts with a valid seed input and generates each fuzz input by randomly selecting a new set of bit locations inside the seed and changing the corresponding bits. The size of this set is specified by a “mutation ratio”, which specifies the mutation size as a proportion of the seed size.

Prior to my initiation of our line of work in fuzzing [8, 2], Black-box Mutational Fuzzing had already gained a reputation of being highly effective and we learned that practitioners often fuzzed each target program using multiple seeds. In fact, the Computer Emergency Response Team Coordination Center (CERT/CC) had released a software suite called “Basic Fuzzing Framework” (BFF) that was developed to optimize the above usage pattern, which is commonly known as “fuzz campaigns”. More precisely, given a target program and a set of seeds, BFF will schedule among the seeds and a set of hardcoded mutation ratios across fuzz runs. Its goal is to increase the number of unique bugs found in a campaign. BFF models the outcomes of fuzz runs with a fixed seed-ratio pair as independent Bernoulli trials, where a success is defined to be the discovery of a new bug. In essence, BFF gradually learns the success rate of each seed-ratio pair, while simultaneously favoring seed-ratio pairs with higher observed success rates thus far. Observing this very notion of exploration vs. exploitation, which is the hallmark of Multi-Armed Bandit (MAB) problems, the authors of BFF deployed an existing MAB algorithm called Upper Confidence Bound as part of the scheduling algorithm in BFF.

My research has identified two theoretical deficiencies in BFF. First, the outcomes of fuzz runs with a fixed configuration are not independent and the success rate is not constant and decays over time. This is evident by observing that the number of unique bugs discoverable when fuzzing with a fixed configuration must be a constant, and that the bugs encountered over time will repeat and thus no longer count as successes. I reasoned the correct way to model such outcomes is the Weighted Coupon Collector Problem (WCCP), with the complication that the weights have to be gradually learned in our setting. Second, I observed that BFF does not consider the time consumed by each fuzz run, which can lead to suboptimal schedules. This is because the expected increase in the number of unique bugs found with a configuration depends on both its current success probability and the time required to perform the next fuzz run with it. Armed with these two insights, my collaborators and I designed several new MAB algorithms for WCCP with unknown weights and evaluated them on 100 real-world programs. In our experiments, one of our
algorithms outperformed BFF by finding 1.5 times more unique bugs in a 10-day fuzz campaign, demonstrating a vast improvement in the state of the art. A major ingredient of this algorithm is also one of my earlier theoretical contributions to our field, where I showed how to use confidence intervals to estimate the probability of finding a new bug in the next fuzz run. This allows our algorithm to allocate more time towards configurations that are currently more promising, which ultimately leads to finding more unique bugs using the same amount of time as BFF.

Besides becoming part of the Ph.D. thesis of one of my collaborators, our first foray into Black-box Mutational Fuzzing has also paved way for two follow-up papers in our research group, both published at top-tier security conferences. I was fortunate to have contributed significantly to one of them, which investigated the mathematics of how to estimate an optimal mutation ratio given a program and a seed [2]. Our experiments showed that our mathematical modeling is highly effective—using mutation ratios estimated by our model, we found 78% of all bugs discovered with empirically optimal ratios in the same amount of time, where the empirical optimal ratios are computed by brute-forcing 1000 ratios spaced 0.001 apart. In addition, our estimated ratios were also often very close to the empirical optimal ratios.

As part of my ongoing work in this area, I have started analyzing a more sophisticated variant of fuzzing known as Grammar-Based Fuzzing. Intuitively, instead of mutating a seed, this variant generates fuzz inputs by sampling from an input grammar. Such fuzz inputs are by definition grammatically valid, but they can be semantically invalid and thus trigger logical bugs in the target program. Due to this nature, Grammar-Based Fuzzing has found significant security applications in language interpreters, compilers, and web browsers. The research questions I am tackling include designing efficient heuristics to approximate known but slow uniform sampling algorithms for context-free grammars as well as evaluating their effectiveness in real-world vulnerability discovery tasks.

**Binary Analysis.** Another major thrust of my research is in Binary Analysis. Broadly speaking, Binary Analysis refers to the use of program analysis techniques towards program binaries, also commonly known as “executables”. As an example security application, we may use static analysis—traditionally an analysis technique on source code—to obtain an over-approximation of the behavior of a given binary and then analyze this over-approximation to rule out classes of malice such as integer overflows.

Although the principles of many source-based techniques are relatively well-understood, applying these techniques to binaries have frequently proved to be challenging in both accuracy and scalability. One of the major reasons is because program binaries do not explicitly represent common abstractions found in program source. As one example, instead of variables declared with types, we see only untyped memory locations and registers in a program binary. This may negatively affect analysis accuracy due to type ambiguities of stored values. As another example, instead of clearly-marked functions, we see only sequences of assembly instructions. This may hinder divide-and-conquer strategies that achieve efficiency by limiting the scope of an analysis to within each function.

My research in Binary Analysis directly addresses these challenges by devising new algorithms to recover lost abstractions to the extent possible. I unify my work in this area under a notion of decompilation—given an input binary, emit high-level C source code that compiles to a binary with identical behavior. In [7], my collaborators and I tackled decompilation head-on by developing a new end-to-end decompiler. Our goal was to emit correct decompiled output with as few gotos as possible, thus recovering the high-level control structure. We started with a structural analysis algorithm originating in the compiler optimization literature and combined it with a novel iterative refinement process. Intuitively, structural analysis repeatedly contracts an input control-flow graph by reducing subgraphs that match control structures such as if-then-else and while-loops; once no matching subgraphs can be found, it will emit the remaining graph using gotos. Our variant with iterative refinement instead emits one carefully-selected goto in the latter case to remove one arc from the graph, and then restarts the contraction phase since the graph may now contain matching subgraphs. In our experiment, we observed a 30-times reduction in the number of emitted gotos with this technique, demonstrating its great utility towards emitting structured output.
Our experience in building a decompiler has informed us of many exciting opportunities. For example, we learned there was a real need to improve our capability to identify all functions in a program binary. This is because an unidentified function will not be decompiled, which will automatically result in an incorrect decompilation output. Recognizing that compiled functions tend to start with certain patterns, my collaborators and I designed a simple supervised machine-learning algorithm for this task [4]. Our evaluation showed that our algorithm outperformed all previous solutions, including the de facto industry standard disassembler IDA Pro. As part of my ongoing work in this area, I am investigating the power of supervised-learning in other abstraction recovery tasks. In particular, I have made substantial progress towards understanding how higher-order Markov Chains can be trained to recognize code-data transitions in a binary, thus avoiding decompilation errors induced by treating data as code. Interestingly, this technique can also detect cases where code is disguised as data, which is a common challenge in malware analysis.

Other Research. Previously I have also worked on two other topics in malware analysis, with the first being static lineage inference [6]. In Biology, lineage inference is the problem of deducing the evolutionary relationships among species. It is highly applicable to malware analysis because many malware strains in the real-world are derived from existing strains. Inferring malware lineage allows us to increase analysis reuse among closely-related strains, which ultimately reduces analysis cost. Our system statically extracted lineage relationships with high accuracy in our experiments, which include a rare instance where we were provided with the true development history of several malware families.

In later work, my collaborators and I investigated a common challenge in analysis reuse [5]. Specifically, we investigated how to build a database of previously-analyzed functions to quickly and accurately retrieve analysis results associated with similar-behaving variants of a given function. As part of our project, we developed a novel function search engine based on dynamic similarity tests—we execute each function in a number of randomized sandbox environments, record distinguishing features on its behavior, and index/retrieve functions using those features. Our engine was effective even when binaries compiled with different compilers and different optimization levels were in the mix. In our experiments, it ranked the correct matches among the top-10 results over 77% of time.

Finally, I have also recently ventured into the hot area of studying the security of software running on commercial off-the-shelf (COTS) devices. Together with my collaborators, we collected over 23,000 firmware images from 42 vendors and built an automated system to run and analyze these firmware images [1]. Since we emulate the target devices in software, our system is highly economical and can scale horizontally with ease. Using our system, we discovered 887 firmware images spanning at least 89 distinct products that are vulnerable to known exploits. Our findings include 14 previously-unknown vulnerabilities, all of which have been disclosed to the vendors responsibly.

Future Research Directions. My long-term research interest is to design cost-effective mechanisms that improve the security of our computing environments. Below I will sketch two research directions. Both of them feature identified immediate tasks and also bear sufficient potential as long-term research.

Characterize Fuzzing and Its Uncertainty. With its ease of deployment and high effectiveness against software developed with current predominant technologies, I believe fuzzing will remain a major vulnerability discovery technique for the foreseeable future. Although entire trade books have been written on the practice of fuzzing, to the best of my knowledge there has been little work to formally study the effectiveness of fuzzing prior to our work [8, 2]. With suitable definitions and supporting software, I believe fuzzing can also be used by defenders to rigorously estimate the uncertainty of undiscovered vulnerability as part of risk management. In fact, since 2015 several news reports have pointed to efforts on Underwriters Laboratories-type certification on the security of connected products/Internet of Things (IoT). I believe studying the characterization of fuzzing and its uncertainty is both timely and crucial for such missions when fuzzing is used as part of the certification process. As an immediate task, I will investigate the characterization of Grammar-Based fuzzing. As a long-term goal, I propose investigating other com-
monly-used fuzzing variants as well as designing new variants that are more effective in practice and more amendable to rigorous analysis.

*Analyze SMT Solving Time and Coping Strategies.* In the last decade, Satisfiability modulo theories (SMT) solvers have become a key enabling technology in many software security analyses, several of which were described in [3]. The running time of these analyses is often dominated by SMT solving. Unfortunately, so far there has been no robust framework that provides a useful complexity measure on an SMT query that can be used to compute an asymptotic or concrete upperbound on its solving time. This severely limits our ability to design cost-effective security analyses that use such upperbounds to maximize their objectives within a given time budget. In the short term, I will attempt to overcome this obstacle using an emergent algorithmic notion known as “empirical hardness measures”, which uses supervised machine learning to build a concrete running time predictor for new input instances. In the long term, I believe this project will provide valuable insight on important input features that should be considered as part of an SMT solving time analysis framework, as well as on strategies to cope with concrete solving times that tend to span many orders of magnitude within a single security analysis.

**References**


