Research Statement

Sang Kil Cha
sangkilc@cmu.edu

Many computer security problems stem from buggy software. Attackers exploit software bugs to infiltrate systems and install malicious software (malware) such as viruses. Therefore, it is imperative to find bugs before an attacker can exploit them and to detect malware prior to infection.

The primary goal of my research is to assure software security. I analyze programs to find bugs and to detect malicious functionalities; study malware and offensive techniques in order to develop defensive mechanisms; and engineer software that is secure and reliable.

My Approach to Research

Security is a rapidly evolving area where there is a battle between malicious and benign parties. Attackers develop a new weapon (exploitation). Defenders then create new protection mechanisms to counter the exploitation. Nevertheless, attackers find new vulnerabilities and break the defenses again. This cycle is often repeated.

My vision is to ultimately break the vicious circle. One may argue that we can prevent attacks by designing a provably secure system ab initio, e.g., by writing a program using a type safe language. However, most theoretically secure methods are often based on assumptions that can be invalidated in practice. For example, the type safety may guarantee the memory safety, but does not prevent all exploitations such as SQL injection and information leaks.

Therefore, I play both sides in order to anticipate and prevent the next generation of attacks. Specifically, I not only design defense systems and implement robust software, but I also actively research novel offensive techniques. By switching places with attackers, I often end up making new discoveries, which help me build systems that are more difficult to exploit. For instance, my colleagues and I designed a system for determining the security relevance of bugs by automatically exploiting them [1, 7], which involves generating an actual attack payload. The technique started as an attack, but developed into a technique fostering software security and reliability.

Past Research

Bug Finding. One of my primary research interests is finding bugs, where the analysis is constrained by a specified resource. Resources appear in many forms such as the total time budget for analyses. A key observation is that when resources are constrained, probabilistic bug finding strategies become akin to gambling in which players wager their resources on bug finding machines. Each machine in the game has a distinct parameter configuration and outputs a random reward (bug) from a distribution for a given time. Therefore, the goal is to allocate resources strategically to each machine in order to maximize the outcome.

We modeled the gambling process as an instance of the Multi-Armed Bandit (MAB) problem where a gambler must deal with the trade-off between exploration and exploitation. Unlike the traditional MAB, however, each machine exhibits diminishing returns: the number of new bugs that each machine can find decreases over time. Therefore, we designed an algorithm that dynamically estimates the reward distribution of each machine. The key intuition is that any randomized bug finding allows us to model the distribution of rewards probabilistically from the history of bug finding trials. We designed a practical MAB algorithm using the intuition, which allows us to find 91 more bugs in 100 applications compared to state-of-the-art methodologies [9].

While our MAB algorithm benefits bug finding, it is not directly applicable in many cases when the parameter space is intractably large. For instance, a seed parameter, which specifies a well-formed input to a program under test, can potentially be any arbitrary string. Therefore, we devised a methodology that effectively reduces the number of seeds to consider. The key insight is that it is more likely to find novel bugs in a program when we cover new code than when we explore the same part of the program again. Therefore, a set of seeds that maximizes the code coverage is likely to augment the number of unique bugs found. We modeled this problem as an approximated minimal set-cover problem. We used our algorithm to accumulate seeds to run a tool, and found 46 more bugs in 10 programs than with the current seed selection algorithms [8].

We also demonstrated the benefits of constraining the search space in symbolic execution—another popular program analysis technique for finding bugs. The core idea is to compromise the completeness of symbolic execution by concentrating our resource on execution paths that are more likely to be vulnerable, e.g., paths that are likely to have buffer overflows. The proposed technique, called *preconditioned symbolic execution* [1, 7], runs the traditional symbolic execution with an additional precondition that prunes away program paths that do not satisfy it. As a result, our technique reduced the time to find security critical bugs by $1.8\times$, and found $2\times$ more bugs given a time limit compared to the current symbolic executors. Our latest symbolic execution system `MergePoint` [3] successfully found 11,687 bugs in total from 33,248 distinct program executables extracted from Debian Linux.

Bug Prioritization. An immediate research question that arises from bug finding is how to prioritize fixing bugs that we found given limited resources, i.e., out of 11,687 bugs that we found from Debian, which should be fixed first? We noted that not all bugs are equivalent. Some bugs lead to critical security breaches, but some others are just aesthetic bugs that simply annoy users. Thus, we developed a technique to prioritize bugs based on the security relevance.

As the first step toward the problem, we demonstrated an *automatic exploit generation* (AEG) technique that verifies whether a given bug is exploitable [1, 2, 7]. To this end, we augment typical safety properties in symbolic execution with an exploitability property—which describes the position of attack code, and the value of overwritten addresses, and so forth—and find a program path where the exploitability property holds. Our analysis is sound, thus, the exploitable test cases generated by AEG lead to a control-hijack attack. Our results on 30 realistic applications on both Windows and Linux showed that AEG is a promising technique for prioritizing software bugs.

Malware Analysis. My research interests include analyzing existing malware to understand its characteristics and to prevent further damage. Note that the number of malware is expanding rapidly, e.g., McAfee catalogs $100,000$ new distinct samples per day in 2013. Unfortunately, current signature-based malware detection is bottlenecked by the number of signatures. We observed that having more signatures not only increases the memory use, but also decreases the speed of signature matching due to cache misses. We assessed the issue of handling the large number of signatures given limited memory with a malware detection system, called SplitScreen [4, 6].

The crux of the work is to split the detection process into two, using a new data structure based upon Bloom filters that have one-sided error: it may occasionally say a program is malicious even when it is benign, but never say malware to be clean. In the first stage, SplitScreen identifies programs that are possibly malicious, and signatures that may match the programs. The second stage performs an exact pattern matching using the subset of signatures and programs identified during the first phase in order to determine precisely which programs are indeed infected. We showed that SplitScreen is $2\times$ faster than the current Anti-Virus (AV) methods, and uses half of the memory. As the number of AV signatures grows, the speedup of SplitScreen will only improve.

Additionally, my research also includes studying offensive techniques. As such, we developed execution-based steganography [5], which enables bypassing dynamic program analysis such as behavior-based malware analysis by hiding its execution behavior.

Future Research

My underlying research aim is to stay ahead of attackers, and make the world's software secure. As the first step towards the goal, I am currently working on resource-aware bug finding and prioritization. However, there are still many open challenges remaining such as follows. First, typical parameter optimization for bug finding is challenging especially when parameters have potentially infinite distinct values and there is no obvious parameter reduction methodology as in seed selection, e.g., mutation ratio is a continuous number [9]. Second, current bug triaging techniques, which classify test cases into groups, are imprecise: they can have both false-positives and false-negatives. Notice bug triaging is an important step for developing bug finding strategies, because the goal of bug finding is to maximize the number of unique bugs, which can only be estimated by triaging. Finally, the state-of-the-art bug prioritization technique still lacks support for advanced exploitation techniques such as use-after-free exploit and heap spraying.

In the next few year, I plan to approach the problems in two ways. First, I am developing a theory on bug finding to precisely describe the current issues and to develop novel techniques on resource-aware bug finding. Second, many of the problems require not only a principled theory, but also tremendous engineering effort to solve them. Therefore, I, as a practical scientist, will continue designing novel techniques, implementing systems, and evaluating them. In a longer term, I aim to expand the horizon of current bug finding techniques to discover various classes of bugs beyond memory-safety bugs, which can even appear in programs written in safe languages, e.g., information leakage.

References

[1] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *Proceedings of the Network and Distributed System Security Symposium*, pages 283–300, 2011.

[2] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic Exploit Generation. *Communications of the ACM*, 57(2):74–84, Feb. 2014.

[3] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the International Conference on Software Engineering*, pages 1083–1094, 2014.

[4] S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. G. Andersen. SplitScreen: Enabling Efficient, Distributed Malware Detection. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, pages 377–390, 2010.

[5] S. K. Cha, B. Pak, D. Brumley, and R. J. Lipton. Platform-Independent Programs. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 547–558, 2010.

[6] S. K. Cha, I. Moraru, J. Jang, J. Truelove, D. Brumley, and D. G. Andersen. SplitScreen: Enabling Efficient, Distributed Malware Detection. *Journal of Communications and Networks*, 13 (2):187–200, 2011.

[7] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[8] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley. Optimizing Seed Selection for Fuzzing. In *Proceedings of the USENIX Security Symposium*, pages 861–875, 2014.

[9] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling Black-box Mutational Fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 511–522, 2013.