Computer systems are playing essential roles in all aspects of our daily life. Protecting these systems against attacks has been one of the most indispensable tasks. However, due to the ever-increasing demand on performance and functionality, operating systems and popular applications are still built on top of unsafe languages, like C/C++. Consequently, attacks stemmed from memory safety issues have led to billions of dollars lost and will continuously plague end-users.

My research aims to secure computer systems by discovering severe attack vectors and developing comprehensive defenses. My research philosophy is to break up complicated systems into smaller, manageable components; with a systematic understanding of the characteristic of each component, I can identify hidden weaknesses and develop cross-boundary solutions. In my previous work, I divide the program memory into two general dimensions: the *control dimension* that determines the control-flow of the execution, and the *data dimension* that hosts the user-inputs and saves calculation results. With a systematic analysis of each dimension, my work demonstrated that: data-oriented attacks are generic, severe and expressive for attackers to bypass all existing practical defenses [6, 7, 9]; with the information from the data dimension, we can construct comprehensive protections against predominant attacks in the control dimension [2, 5]; user-inputs provide practical guidance to identify necessary functionalities and help customize the code to reduce the attack surface [3, 10].

My research has received the Best Paper Awards from CCS 2019 and ICECCS 2014, and has led to many real-world impacts. TypeDive [2] detects 35 new bugs in the Linux kernel that either leak sensitive information or allow excessive permissions. Data-oriented attacks on Chrome [6] got the acknowledgment from Google and they have implemented site isolation to eliminate the reported threat. I have submitted two proposals as Co-PI. The one on detecting weird machines has been awarded $805,070 dollars by DARPA from January 2020 to June 2021. I led the proposal preparation and my research work [7, 9] on data-oriented attacks become the foundation of this grant. Another proposal on automatically selecting fuzzing parameters is under review. Now I am leading the development of two ONR fundings about program debloating, where Razor [3] is the first outcome of these two projects. Other than software security, my research also spans the following areas of computer science: compilers [2, 4, 5, 7]; binary analysis [8, 9]; systems [1, 6, 12]; trusted computing [10, 11].

Attacks in Program Data Dimension

The salient characteristic of research in system and software security is the active confrontation between defenders and attackers: defenders analyze bugs and attacks to find common patterns, and build practical solutions to mitigate the problem; attackers keep inspecting the protected system to find the weakest points, and develop new techniques to revive their attacks. With the recent deployment of defense mechanisms, like control-flow integrity (CFI), existing attacks that divert the program control-flow get more and more difficult to compromise vulnerable systems. However, since attackers will never stop constructing new attacks, the remaining question is: *what is the next generic attack vector?*

To answer this question, I systematically examined the program code dimension and data dimension, and found that data-oriented attacks, which do not directly change the control-flow, could be the next attack target. Specifically, my research demonstrates that data-oriented attacks are as generic, expressive, severe and easy to construct as existing control-flow attacks. We should put more effort into developing defense mechanisms to prevent these attacks.

Expressiveness. I proposed *data-oriented programming (DOP)*, a new general methodology to build expressive data-oriented attacks from traditional memory safety issues [7]. Although previous work showed that data-oriented attacks are possible, the common wisdom believes that this type of attacks heavily rely on particular functions and variables, and thus have limited expressiveness. DOP extends our understanding of data-oriented attacks by showing that without any specific variables or functions, attackers can still construct expressive data-oriented attacks on common vulnerable systems, and even perform Turing-complete computations. Constructing attacks with DOP faces three challenges, as data-oriented attacks do not alter the program control-flow: 1) how to run preferred instructions in a particular order; 2) how to deliver intermediate results across different instructions; 3) how to skip undesired instructions in between.

The idea of DOP is to chain a lot of *data-oriented gadgets* through *gadget dispatcher*s to achieve expected computations. Each data-oriented gadget is a sequence of instructions that load operands from memory, perform semantic operations (e.g., addition), and store the result back into memory. With the vulnerability, we can change the address of each load or store in a desirable way: by changing the load address of one gadget to the store address of another, attackers can effectively deliver the calculation result of the second gadget to the first one; by changing the store address of one gadget to some never-read addresses, the execution will drop the side effect of undesired instructions during the execution. In this way, we can selectively *activate* particular gadgets and chain them together for high-level semantics. In order to run desired data-oriented gadgets in a particular order, we utilize the gadget dispatcher, a loop that contains many diverse data-oriented gadgets, and has a selector (e.g., a memory-error bug) to activate different gadgets in each iteration. During a DOP attack, the program executes over the loop many times, and each time activates some particular gadgets. In this perspective, the vulnerable program is a weird machine that executes instructions (i.e., data-oriented gadgets) beyond the developer's intention, and attackers can *write* code in the paradigm of data-oriented programming to achieve their purpose .

To support the construction of DOP attacks, I implemented a compiler pass to automatically identify data-oriented gadgets and dispatchers from program source code. The evaluation on real-world programs demonstrated the prevalence of DOP elements and confirmed the feasibility to build Turing-complete attacks. Specifically, from nine widely used applications, including web servers and file servers, I identified 7518 data-oriented gadgets and 1443 gadget dispatchers. Eight programs have all gadgets for simulating arbitrary computations and two of them are confirmed to be able to build Turing-complete attacks. I constructed three end-to-end attacks that bypass randomization-based defenses without information leakage, simulate a network bot, and alters the code pages to revive code injection attacks.

Automatic Exploit Generation. Once we know that data-oriented attacks are expressive, the next question is *how to build such attacks?* Since existing work relies on manual effort, I developed FlowStitch [9] to construct data-oriented attacks automatically. The challenge is to identify security-critical variables and to corrupt them by given vulnerabilities. My solution is two-fold: first, FlowStitch identifies security-critical variables from common security-related sinks, like user id in *setuid()*; second, it stitches two disjoint data-flows to build attacks, one of which contains the security-critical data. In the second step, FlowStitch systematically searches all possible connections between two data-flows for feasible stitching. FlowStitch automatically constructs 19 data-oriented attacks against eight vulnerable programs.

Real-world Attacks on Chromium. I applied the new methodology to construct data-oriented attacks against Chromium — the open-source version of Google Chrome browser [6]. After a comprehensive study of Chromium, I found many variables that guard the browser security settings, which may disable the same-origin policy (SOP) or allow access to arbitrary local files, cookies and user's locations. For example, corrupting the flag *m_universalAccess* will disable the SOP check and enable attackers to visit any resources from any web domains. We reported our findings to Google. Recently, Google researchers have developed process-based site isolation, which provides a chance to completely eliminate this security threat.

Protections with Cross-dimension Data

After revealing the severity of data-oriented attacks, I spent some effort to build defenses. I first studied recent defenses in the control dimension. Specifically, control-flow integrity (CFI) aims to block control-flow hijacking attacks. However, even the latest CFI proposal cannot completely block all malicious hijacks of the control-flow. Therefore, before developing defenses against data-oriented attacks, I build solutions to eliminate the threat of control-flow attacks.

My research on enforcing control-flow integrity includes two parts, both borrowing information from the data dimension to assist the protection in the control dimension. First, I proposed the *unique code target (UCT)* property, which will block all control-flow hijacking attacks. I developed the first end-to-end system that enforces the UCT property. Second, I introduced the technique of *multi-layer* type analysis to refine indirect call targets. While the first work uses dynamic information to enforce UCT, the second is a general way to build accurate control-flow graphs.

Dynamically Enforcing the UCT Property. I proposed the UCT property: *each invocation of each indirect-call instruction has one and only one allowed target.* Enforcing UCT is necessary to stop control-flow attacks, as otherwise, attackers can find indirect-calls with multiple targets and divert the control-flow to the unintended ones. To enforce UCT, we have to collect dynamic information to infer the unique target, which brings two challenges. First, it is unclear what runtime information is needed to infer the unique target. Previous work utilized the execution path to distinguish different targets. However, my work shows that even the complete execution trace may still allow hundreds of targets for some indirect-calls. Second, there is no efficient way to collect the dynamic information for target inference and UCT enforcement.

To address these challenges, I designed and implemented μCFI, the first end-to-end system enforcing UCT [5]. During the program compilation, μCFI systematically searches in the program to identify the necessary information for inferring the unique target. For example, if the indirect-call target is retrieved from a table, we cannot tell which element is used from the execution path, unless we know the table index. We call data like table index as *constraining data.* Formal proof shows that if we collect both the execution trace and the constraining data, we can infer the unique target for each indirect-call. To efficiently collect the execution trace, I utilize Intel Processor Tracing (PT) — a hardware feature available on modern Intel CPUs to dump the control-flow information to a protected region. However, Intel PT does not record any constraining data. I implemented software-based instrumentation to encode any data into the control-flow so that Intel PT will record it effectively. From the PT trace, a monitor running in a different process can find the execution path and restore the constraining data. The monitor infers the unique code target for each indirect-call and compares it with the real one used in the execution. We can tell the application is under control-flow hijacking attacks if these two values are different.

We applied μCFI to protect standard performance benchmarks (SPEC CPU2006) and two server programs (Nginx and vsftpd) to evaluate its efficacy and overhead. We also test it against various control-flow hijacking attacks, including five real-world exploits, one proof of concept COOP attack — the most sophisticated, stealthy method to hijack the control-flow, and two synthesized attacks that bypass existing defenses. The results show that μCFI strictly enforces the UCT property

for all protected programs, successfully detects all attacks, and introduces less than 10% performance overhead.

Refining Indirect-call Targets through Static Analyses. Refining indirect-call targets also brings benefits to general static program analyses, like enabling compilers to perform aggressive optimizations. I developed TypeDive [2] to refine the targets of indirect calls through *multi-layer* type analysis. The key observation is that a function address is usually stored into a function pointer with a *multi-layer type*, and before indirect calls, the pointer value is retrieved from the same type. Examples of multi-layer types include nested structures (e.g., A.B.C) and cross-references between structures (e.g., A→B→C). Since the multi-layer type captures more comprehensive information than the single-layer type, TypeDive effectively refines the indirect call targets by matching types of functions and pointers. We applied TypeDive on the Linux kernel, the FreeBSD kernel and the Firefox browser. It successfully eliminates 86% to 98% targets than existing approaches do, and helps find 35 new semantic bugs from the Linux kernel.

User-driven Attack Surface Reduction

Other than building defenses against specific attacks, my research also develops generic tools to reduce the attack surface of existing systems, through program debloating and trusted computing.

Program Debloating Driven by User Input. Modern software contains massive features to satisfy all potential users, but each individual user only requires few of all implemented functionalities. The unused code not only hinders the optimal execution, but also introduces a large attack surface, like exploitable bugs. Eliminating developer-intended, but user-undesired code is an emerging technique, called *program debloating*. The debloating process should take place after software deployment so that we can tailor the program for each user. However, there are two challenges to debloat binaries: how to allow users to conveniently express their requirements and how to remove unnecessary code while keeping expected features.

To solve these problems, I developed Razor [3], a tool for debloating application binaries based on the user requirements. It allows users to provide sample inputs to demonstrate of the expected features, and uses control-flow based heuristics to identify more code related to the required features. First, Razor runs the program with given inputs and records the execution trace, including basic blocks, conditional branches and indirect calls. Second, it rewrites the binary to generate a new one with unexecuted code removed. However, since user-provided inputs usually have limited coverage of the required features, this simple method may remove required-but not-executed instructions from the binary. To solve this problem, Razor takes control-flow-based heuristics to include more related code into the execution trace. The intuition is that, if a neighbor path has a similar control-flow graph as the executed one, it likely implements the similar functionality as the executed code. We developed four control-flow-based heuristics to infer related code. For a given program, Razor gradually increases the heuristic level to include more and more code into the debloated binary, until the generated program is stable.

We applied Razor on commonly used benchmarks and real-world programs, including the web browser Firefox and the closed-source PDF readers FoxitReader. Razor successfully removes over 70% code from the program binary while preserving the required functionalities. At the same time, it introduces only 1.7% overhead to the new binary.

Trusted Computing. Mobile operating systems (e.g., Android and iOS) form large, complex systems and are prone to various vulnerabilities. Meanwhile, users tend to store a lot of personal sensitive information on mobile devices. Protecting such sensitive information is important but challenging. I introduce the concept of the *trusted data vault*, a small trusted engine that securely manages the storage and usage of sensitive data in an untrusted mobile device. DroidVault [10] is the first realization of a trusted data vault on the Android platform. It establishes a secure channel between data owners and data users while allowing data owners to enforce strong control over sensitive data with a minimal trusted computing base (TCB). I prototyped DroidVault via the novel use of the hardware security feature of ARM processors, i.e., TrustZone.

Future Research

Due to the active confrontation in system and software security, attackers will continuously develop new attack vectors and detect new program bugs. Therefore, I will continue my work on building secure systems. In the following several years, I plan to extend the data-oriented attack to make it independent of memory safety issues, and meanwhile, build practical defenses with new hardware primitives. My long term plan is to develop techniques that can automatically infer general, applicable rules from previous security practices to simplify future security research.

Generalization of Data-oriented Attacks. I would like to extend data-oriented attacks to more general attack scenarios, without dependency on memory safety issues. My first observation is that many simple programs, like applications running in IoT (Internet-of-Things) devices, directly use external inputs as internal variables. In this case, we can provide malicious input to manipulate the program data. For example, a heater uses the current temperature to decide to continue working

or not, whereas attackers can provide a wrong value to keep increasing the temperature. I plan to analyze IoT programs to identify critical data that have severe security implication while can be manipulated easily through user input. This work will require program analysis techniques on source code and binary, which I have obtained experiences from previous work.

The second observation is that program implementation usually encodes many extra, unexpected functionalities. For example, the dynamic loader is designed to load shared libraries, but is turned out to support arbitrary executions. Attackers can create malformed inputs to activate the hidden functionalities inside the code, and may use them to launch data-oriented attacks. I plan to build a platform to systematically search inside the program state space to identify hidden behaviors, and utilize them to construct attacks. My previous work has covered techniques for exploring program state space, like fuzzing and symbolic execution, which will be helpful in this work to identify hidden functionalities.

Hardware-assisted Defenses against Data-oriented Attacks. I plan to develop defenses against data-oriented attacks with the help of hardware primitives. Currently, the principled protection, memory safety, have unacceptable overhead to the protected system (usually more than 100%), hindering their adoption. The reason is that memory safety relies on heavy software-level instrumentation to check the integrity of each memory access. My previous work [5] shows that we can utilize hardware features to accelerate the safety check, even if the feature is not originally designed for security. Researchers in industry are actively developing new hardware features. For example, Intel has developed MPX, MPK and CET, while ARM has introduced Pointer Authentication. Several companies have announced their plan to support *tagged memory*, where each memory has a tag indicating its types: code pointer, data pointer or pure data. I plan to utilize the tagged memory to prevent data-oriented attacks. Specifically, we can tag data with sensitive or non-sensitive, and only particular instructions can operate on sensitive data. I will also collaborate with hardware vendors to develop new primitives for security.

Automatic Insight Extraction to Assist Future Security Research. Current security research heavily relies on the insight and experience of individual researchers to identify new problems and build new solutions. Based on my observation, a large number of projects performed by different researchers follow similar methodologies; meanwhile, the experience we learn from one project can be applied to many other projects. Therefore, my long-term plan is to develop techniques that can automatically extract insights from previous projects, and apply these insights to assist future security research.

Bug finding and analysis provides a good example to illustrate my motivation. Traditionally, researchers first manually analyze several vulnerabilities to extract a common pattern, and then write analyses to automatically search the appearances of the pattern to find new bugs. Instead of performing these steps repeatedly for different types of bugs, we should think about 1) how to automatically infer the common pattern for given vulnerabilities; 2) how to automatically write analysis to detect bugs. We may utilize machine learning algorithms to extract common features from known vulnerabilities, where existing projects will provide us a list of insightful candidate features. We may be able to extract a general template of program analysis that can be easily extended to detect different bugs. Even if we cannot fully automate these steps, we should be able to save many manual efforts from the previous experience. Once a vulnerability is detected, we can reuse the knowledge learned from existing vulnerabilities to measure its severity and to construct attacks. For example, if we have learned that one program component has access to high-privileged resources, a new bug in this component will likely have a high severity. If we have built attacks against one program, we can reuse the component of the old attack to help build new attacks, like the security-critical data in data-oriented attacks, or gadget chain in code-reuse attacks.

This work will require both traditional program analysis techniques and emerging machine learning algorithms. My previous work has explored various techniques for program analysis, like source-code analysis and binary-analysis, data-flow analysis and control-flow analysis, fuzzing and symbolic execution, and so on. I will look into machine learning algorithms to automatically summarize commonalities from multiple analysis results and provide helpful insights.

References

[1] Jinho Jung, **Hong Hu**, Joy Arulraj, Taesoo Kim, and Woonhak Kang. Apollo: Automatic Detection and Diagnosis of Performance Bugs in Database Management Systems (to appear). In *Proceedings of the International Conference on Very Large Data Bases (**VLDB**)*, September 2020.

[2] Kangjie Lu and **Hong Hu**. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (**CCS**)*, November 2019. **Best paper award**.

[3] Chenxiong Qian*, Hong Hu*, Mansour Alharthi, Simon Pak Ho Chung, Taesoo Kim, and Wenke Lee. Razor: A Framework for Post-deployment Software Debloating. In *Proceedings of the USENIX Security Symposium (**Security**)*, August 2019. * **Co-first authors.**

[4] Jinho Jung, **Hong Hu**, David Solodukhin, Daniel Pagan, Kyu Hyung Lee, and Taesoo Kim. Fuzzification: Anti-Fuzzing Techniques. In *Proceedings of the USENIX Security Symposium (**Security**)*, August 2019.

[5] **Hong Hu**, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the ACM Conference on Computer and Communications Security (**CCS**)*, October 2018.

[6] Yaoqi Jia, Zheng Leong Chua, **Hong Hu**, Shuo Chen, Prateek Saxena, and Zhenkai Liang. The "Web/Local" Boundary Is Fuzzy - A Security Study of Chrome's Process-based Sandboxing. In *Proceedings of the ACM Conference on Computer and Communications Security (**CCS**)*, October 2016.

[7] **Hong Hu**, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *Proceedings of the IEEE Symposium on Security and Privacy (**Oakland**)*, May 2016.

[8] **Hong Hu**, Zheng Leong Chua, Zhenkai Liang, and Prateek Saxena. Identifying Arbitrary Memory Access Vulnerabilities in Privilege-Separated Software. In *Proceedings of the European Symposium on Research in Computer Security (**ESORICS**)*, September 2015.

[9] **Hong Hu**, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic Generation of Data-Oriented Exploits. In *Proceedings of the USENIX Security Symposium (**Security**)*, August 2015.

[10] Xiaolei Li, **Hong Hu**, Guangdong Bai, Yaoqi Jia, Zhenkai Liang, and Prateek Saxena. DroidVault: A Trusted Data Vault for Android Devices. In *Proceedings of the International Conference on Engineering of Complex Computer Systems (**ICECCS**)*, August 2014. **Best paper award**.

[11] Li Li, **Hong Hu**, Jun Sun, Yang Liu, and Jin Song Dong. Practical Analysis Framework for Software-based Attestation Scheme. In *Proceedings of the International Conference on Formal Engineering Methods (**ICFEM**)*, November 2014.

[12] Xinshu Dong, **Hong Hu**, Zhenkai Liang, and Prateek Saxena. A Quantitative Evaluation of Privilege Separation in Web Browser Designs. In *Proceedings of the European Symposium on Research in Computer Security (**ESORICS**)*, September 2013.