Research Statement

William Robertson
Northeastern University

Introduction

Data and computation has become inextricably intertwined with modern society. Yet, the systems, networks, and software that form the technological substrate upon which modern society functions is unworthy of our trust. New security vulnerabilities are discovered at an ever-increasing rate, outpacing our ability to remove them or defend against their exploitation. At the same time, we must simultaneously acknowledge that the ability to find all vulnerabilities or prevent their introduction into production systems currently lies far beyond our grasp. This “trust gap” is a critical problem that will only grow in magnitude if our current trajectory remains unchanged.

My research focuses on closing this gap. While I have many interests that span a variety of domains, my research agenda has been guided by a three-fold approach:

(1) anticipating and measuring novel threats,
(2) developing new techniques for discovering vulnerabilities, and
(3) building systems that defend against entire vulnerability classes by design.

By anticipating and measuring security threats, we can proactively defend against emerging attacks on a data-driven, quantifiable basis rather than relying on a predominantly reactive approach to defending systems. By developing techniques to discover vulnerabilities that heretofore were invisible, or scaling our ability to do so, we can more effectively harden systems before they are exposed to attackers. Last but not least, building on insights drawn from measurement and vulnerability analysis enables the creation of new systems that are inherently secure against known threats.

Post-tenure, I have been afforded the opportunity to take a step back and contemplate the next phase of my research career and, in particular, the most pressing security problems I would like to solve. Thus, while I will summarize and put into context significant examples of the work I have performed to date, I will also describe several major research directions that I plan to pursue in the next epoch of my career.

Measuring Security

The genesis of many of my research projects lies in measurement: data collection, hypothesis testing, and knowledge synthesis. In the following, I will discuss my measurement efforts in two areas: web security and emerging malware.

Web Security

A significant advance in client-side web security was the introduction of Content Security Policy (CSP) in modern browsers. CSP enables web developers to specify richer security policies than the rigid same-origin policy that has been the default security policy since the web’s inception. However, despite its promise anecdotal evidence began to emerge that CSP adoption was far less enthusiastic than anticipated. In *Why is CSP Failing? Trends and Challenges in CSP Adoption* [19], my group reported on a long-term, large-scale study of CSP usage on the open web. We crawled the Alexa Top 1M on a weekly basis for over a year, and found that CSP was deployed in enforcement mode on only 1% of the Alexa Top 100. Adoption precipitously dropped off at lower Alexa ranks. We also studied why organizations were reluctant to deploy CSP, or why those that did deploy CSP did so in ineffective ways, and identified several usability concerns. Some of these concerns, such as the inability to securely include inline script, were addressed in a subsequent version of CSP. Others, such as the difficulty of defining tight policies for highly dynamic web sites, remain a significant barrier to CSP adoption to this day as reported by follow-on work from other research groups.

Web applications often heavily depend on third-party resources that includes hosted JavaScript, inducing a complicated and sometimes highly dynamic trust graph. In *Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web* [12], my group examined the implications of this in the context of outdated JavaScript containing known vulnerabilities. In this study, we crawled 133K popular web sites from the Alexa rankings, and found that fully 37% of the measurement corpus included at least one third-party JavaScript library with a known vulnerability. Furthermore, we found that transitive code inclusions – that is, direct inclusions that in turn perform their own inclusions – as well as ad-tracking code were more likely to be vulnerable. This work helped to highlight the haphazard manner in which dependencies are managed in web applications, and helped to encourage more systematic and careful dependency management frameworks for web applications.

Recently, my group has begun to examine emerging security issues in complex web applications with distributed caching architectures. In "Cached and Confused: Web Cache Deception in the Wild" [15], we measured the prevalence of a particular form of cache vulnerability – web cache deception – in which a fronting cache erroneously caches private user data. If an attacker is able to trick a user into visiting a URL that results in this data being cached, the attacker can then later access that cache entry, obtaining the corresponding private data. Our work found that 340 sites within the Alexa Top 5K that used popular content distribution networks were trivially vulnerable to web cache deception. Furthermore, we strongly suspect that this number significantly undercounts actual prevalence, since our measurement techniques were necessarily conservative in order to minimize harm to production sites and real users. This work highlights that modern web applications are highly complex distributed systems with security-relevant dependencies that are often not considered. This insight motivates current and future work in my group in the web and cloud security spaces.

Emerging Malware

A second area of interest when it comes to security measurement is emerging malware. Malware continues to plague the Internet despite the best efforts and billions of dollars of investment in the software security industry. One recent example of an emergent trend in malware activity is the rise of ransomware, or malware that holds computational resources or user data hostage in exchange for payment of a ransom. In *Unveil: A Large-Scale, Automated Approach to Detecting Ransomware* [10], my group published a methodology and system for identifying novel ransomware variants at scale. In particular, we demonstrated that dynamic analysis using traditional malware sandbox platforms can in fact be an effective approach for ransomware detection when composed with

(1) environmental manipulation that elicits ransomware behavior from generic malware samples, and
(2) detection heuristics over streams of file system operations tuned towards patterns that encode invariants of ransomware behavior.

Our evaluation demonstrated that Unveil was able to efficiently and accurately identify and classify known ransomware samples. In addition, it was also able to identify a new ransomware family that was previously unknown to the security community.

Malware is constantly evolving to adapt to defenses and exploit new weaknesses. Online survey scams was another recent emerging trend in malware that our group considered in *Surveylance: Automatically Detecting Online Survey Scams* [9]. Our work was the first to systematically highlight and identify the survey scam ecosystem. Our detection algorithm was shown to be effective in identifying more than 8K sites involved in presenting scams, collecting private information, and distributing various forms of malware.

Vulnerability Discovery

A second body of research I have conducted concerns vulnerability discovery. This work can be classified as belonging to one of two broad categories: mixed static-concolic program analyses and dynamic testing to identify targeted classes of vulnerabilities.

Static and Concolic Program Analysis

Mobile devices and applications have been an area of interest for my research group over the past ten years. One observation that became apparent after sustained research into Android malware is that as attacks move towards higher abstraction layers and away from lower-level attacks such as memory corruption, they tend to become less general. That is, attacks more often rely on violating *application-specific* security properties that are also virtually never formally stated. Without formal statements of these properties or the intended behavior of the application, it is seemingly impossible to reliably detect such malware with general techniques.

However, we also observed that subtle Android malware is also often triggered under very specific environmental circumstances in order to avoid detection during testing or sandbox analysis. In *Towards Detecting Logic Bombs in Android Applications* [8], we developed a program analysis that combined concolic execution with static control dependency analysis to identify potential triggers for suspicious security-relevant behavior. Our evaluation over a large corpus of Google Play Store applications supported the ability of our prototype, called *TriggerScope*, to identify both benign and malicious triggered behavior. This ability was further supported by TriggerScope's ability to identify triggered behavior in DARPA Red Team-authored malware as well as real malware samples such as HackingTeam's RCSAndroid family.

My group's work on mobile application security later considered unintended information disclosure vulnerabilities that can arise in interactions with cloud APIs. In *Semi-Automated Discovery of Server-based Information Oversharing Vulnerabilities in Android Applications* [11], we developed a novel program analysis that identifies fields in data objects returned from cloud APIs that are never displayed in mobile application user interfaces. These fields represent potential unintended information disclosures; however, to confirm vulnerabilities and thereby reduce the static analysis's false positive rate, we then applied a dynamic testing step. Our evaluation over a large corpus of Android applications demonstrated the effectiveness of this two-phase mixed static-dynamic analysis by discovering several previously-unknown oversharing vulnerabilities in eight applications.

Dynamic Testing

A significant portion of my work has involved novel dynamic testing approaches for uncovering classes of vulnerabilities. In the past, I have applied dynamic testing to a variety of domains, from detecting access control vulnerabilities in graphical user interfaces [16] down to frameworks for systematically testing CPUs for exploitable microarchitectural flaws [13]. One particularly influential publication in this area is *LAVA: Large-scale Automated Vulnerability Addition* [6], which considers the problem of *evaluating* vulnerability detection tools and techniques that rely in part upon dynamic testing. Evaluations of such tools have historically suffered from a scarcity of test corpora of sufficient scale and diversity. Building a single corpus is time-consuming and error-prone. LAVA proposes synthetic bug injection as an alternative for generating evaluation corpora that ship with both ground truth as well as proofs of vulnerability. LAVA has become a widely-used standard benchmark in fuzzer evaluations, and I remain interested in further research to improve synthetic bug generation.

A recent focus of my lab was on algorithmic denial-of-service vulnerabilities, where attackers can craft malicious inputs to software that induces worst-case behavior and availability failures. In *HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing* [1], my group proposed a new dynamic testing technique for exposing algorithmic denial-of-service vulnerabilities located deep inside Java code. Building in Godefroid's concept of micro-execution, we propose *micro-fuzzing*, which optimistically fuzzes Java methods without having to drive a program's execution to those methods. To enable this, we synthesize program environments – e.g., method arguments and global variables – that represent realistic inputs to these methods as gathered from recorded execution traces. Fuzzing trials are executed on a specially-modified JVM that records time and memory profiles. Those trials that exceed baseline time or memory thresholds are then automatically synthesized as whole-program test cases that are run on a commodity JVM. If the test case reproduces the vulnerability, then the vulnerability is considered to be confirmed. HotFuzz was evaluated on a large number of real Java libraries as well as the DARPA STAC test corpus, and found a number of algorithmic complexity vulnerabilities across the entire test corpus.

Designing Secure Systems

The third broad approach my research embodies is secure systems design that builds on insights drawn from measurement and vulnerability discovery. In the following, I will focus on significant papers related to application hardening and user privacy.

Application Hardening

My group's work on CSP and other web security technologies highlighted that client-side code was becoming an increasingly security-relevant component of web applications. In *ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities* [20], we investigated an instrumentation-based approach to prevent the exploitation of latent vulnerabilities in browser-hosted JavaScript. This instrumentation encompassed two phases: a learning phase that made use of Daikon invariant inference on function arguments, and an run-time phase that enforced these invariants in deployed programs. Our prototype, built on the Closure compiler, was able to achieve this without resorting to annotations or heavyweight training. Our performance evaluation also demonstrated that ZigZag's security guarantees are obtainable for real-world, JavaScript-heavy web applications with modest overhead.

Another research direction that I remain intensely interested in is efficient, low-friction least privilege application separation. In *Trellis: Privilege Separation for Multi-User Applications Made Easy* [14], my group proposed an access control technique for robustly partitioning data in multi-user applications. Trellis modifies the kernel and system libraries to enforce an access control lattice for sections of code and data within an application. Applications can then use a special API to partition applications according to the sensitivity of code and data. A modified LLVM toolchain then converts this partitioning into an efficient binary representation for enforcement. Our case studies demonstrated that with minimal developer effort, a Trellis-enabled system can provide strong isolation guarantees between different intra-process privilege levels with low run-time overhead.

Privacy

The second area of secure systems design I will present here concerns user privacy. One prominent example of this research thrust is my work on defending user privacy within the operating system. Many users clearly care about the privacy of their data and online behaviors, as evidenced by the widespread proliferation and usage of privacy defenses—e.g., network-based anonymity services such as Tor. On the client, full-disk encryption (FDE) is commonly provided by the operating system in order to safeguard user data at rest. This, however, has the drawback that users can be forcibly coerced into divulging secrets —i.e., FDE passwords—that expose otherwise protected information. Another approach to this problem is application-specific privacy modes, such as private browsing or incognito mode as found in modern web browsers such as Chrome, Firefox, and Safari. In this model, users can browse the web with the expectation that no traces of their browsing session will persist to the system in the form of cache entries, history, or bookmarks. This privacy model is of interest as it promises that no information about the browsing session can be recovered from the system through coercion or otherwise. However, research has shown that application-specific privacy modes are difficult (or impossible) to develop correctly, and there is the additional drawback that significant effort must be invested to implement privacy modes on a per-application basis.

Therefore, my group developed PrivExec [17], a technique for providing a system-wide notion of *private execution* to all applications. The technique modifies the system to create two classes of processes: *public* processes that operate with traditional semantics, and *private* processes that are behaviorally restricted to enforce a forensic standard of user privacy. In particular, private processes—or, more precisely, private process groups—are confined to a private storage namespace that is an encrypted read-write overlay on top of the public file system. In addition, the system swap space is partitioned and encrypted on a per-private process group basis. These encrypted storage spaces are secured by a randomly-generated private execution key (PEK) that is stored within the kernel and never released to the user. When a private execution session terminates, the associated PEK is securely erased from memory, implementing cryptographic erasure of the private session data. The end result is a system where any application can enter a private execution mode with strong, cryptographically-backed guarantees of data confidentiality, and users cannot be coerced into revealing data after the fact as they have no knowledge of the corresponding private execution key. Our prototype implementation for Linux demonstrated that strong private execution could be achieved in an efficient manner with minimally invasive modifications to the operating system kernel.

My group later considered the related problem of secure deletion of data from non-volatile storage. Storage devices such as SSDs make use of techniques like wear leveling that involve hidden replication of data, which lies in direct opposition to efforts to securely delete data from storage. In *Eraser: Your Data Won't Be Back* [18], we again built on cryptographic erasure to transparently encrypt files on an insecure medium with a file-specific key. Under this system, erasure is accomplished not by deleting the data but rather securely discarding the key. In order to render the approach tractable, keys are organized in an efficient tree structure where a single master root key is confined to a secure store that is unlocked at boot by the user. Our prototype implementation demonstrated that this approach can be achieved with overhead comparable to popular full disk encryption implementations.

Current and Future Work

My work to date has considered a broad range of security issues and domains. Post-tenure, I have decided to more tightly focus my efforts on several security problems I consider of particular importance. In the following, I will briefly describe these research directions.

Scaling Program Testing

Fuzz testing has become the predominant method by which new vulnerabilities are discovered. Large-scale efforts such as Google's ClusterFuzz, OSS-Fuzz, and FuzzBench are have shown that fuzzing can be highly effective for preemptively discovering and removing vulnerabilities before they are exposed to adversarial input. However, despite intense interest from the academic and industrial security communities, grey-box mutational fuzzing remains fundamentally limited by its inability achieve anything even approaching 100% code coverage. Recent research, including our own work which involved close to 1M CPU-hours of fuzz testing [5], has demonstrated that the current state-of-the-art quickly saturates paths that are easy to cover but inevitably fails to make progress in covering the remainder of the program. In fact, there is readily apparent transition over time from linear to exponential difficulty in covering new code.

The research questions I want to answer in this area center around understanding and breaking through this "coverage wall." In particular, my current work focuses on understanding why some paths are more difficult to cover than others by recording and characterizing path constraints. Given a general predictive model for path coverage difficulty, we can potentially better design mutation and seed scheduling algorithms to balance between the opposing optimization objectives of breadth versus depth and thereby substantially improve program defect discovery rates.

Rehosting Embedded Systems

A second area of interest is embedded systems, which encompasses a wide range of computational devices across the Internet of Things and industrial control systems. Due to several factors, embedded systems are considered to contain more vulnerabilities than traditional systems. However, due to resource constraints it is generally infeasible to apply state-of-the-art vulnerability discovery tools and techniques on embedded systems. In response, there is great interest in transplanting, or *rehosting*, firmware extracted from embedded systems into analysis environments with sufficient resources and introspection capabilities to carry out security analyses such as fuzzing campaigns.

However, while rehosting is enticing, several fundamental challenges remain unsolved. A well-known example is the lack of peripheral hardware models, which is greatly exacerbated by the enormous diversity of SoCs and associated custom hardware available in the embedded market. Aside from this, rehosting is currently an intensely manual process that requires extensive specialized expertise to carry out what we term the *rehosting loop*: the process of booting a rehosted system, debugging failures, identifying the root cause of errors, and modifying the rehosting environment to correct or avoid those errors [7]. My current research in this area lies first in automatically synthesizing usable models of underspecified peripheral hardware from recorded execution traces, and second in accelerating the rehosting loop through *whole-system slicing* to scope rehosting efforts, define fidelity metrics and success states, and automatically identify and avoid error states.

Software Hardening

The third major area of interest I will discuss is software hardening. This effort builds on my group's past work [14], with the goal of devising novel targeted hardware security primitives that can be leveraged by operating system, run-time loader, and compiler toolchain features to provide strong security guarantees for next-generation software. As one example of work in progress, we are building on memory protection key enhancements to RISC-V to enable fine-grain confinement of third-party code. Through enhancements to Clang and LLVM, we are able to easily define intra-process protection domains and least-privilege policies for third-party code that are enforced at run-time via novel OS security features backed by hardware guarantees. This work is aimed squarely at defending against software supply chain vulnerabilities, preventing exfiltration of sensitive data or attacks on computational integrity via trustworthy software partitioning.

To validate this approach, my group has in parallel investigated incarnations of these software hardening techniques within the limitations of existing primitives. One line of work has focused on Intel Memory Protection Keys (MPK), alternatively known as Protection Keys for Userspace (PKU). MPK is a security extension to the Intel ISA that allows developers to partition an address space into distinct page-granularity domains tagged with one of 16 keys. MPK designates bits 59–62 of each page table entry (PTE) as that page's protection key. Thus, MPK is a form of tagged memory, a venerable architectural security approach that dates back to 1960s-era LISP machines and Burroughs mainframes. MPK additionally adds a protection key rights register for user pages (PKRU) to each CPU thread. At every memory access, the MMU checks whether the corresponding PTE's protection key is present in the current thread's PKRU register – either reading or writing can be separately permitted. If so, the access is allowed and execution continues; otherwise, a hardware exception is raised. The PKRU can be updated using a special instruction (`wrpkru`), making domain switches relatively efficient. Also, since the MMU enforces access policies, domain checks are also efficient, incurring essentially zero overhead.

MPKAlloc [2] builds on MPK to isolate allocator metadata from the rest of a program, effectively partitioning a program into a trusted allocator and an untrusted, potentially vulnerable, program. The key invariant MPKAlloc preserves is that any access made to allocator metadata must originate from a trusted domain, and the trusted domain only contains allocator code. By default, all CPU threads execute within an untrusted domain. Upon any allocator invocation – e.g., to allocate or free a heap chunk – MPKAlloc switches the thread to the privileged domain to enable metadata access. Once the requested allocator operation is performed, the CPU thread's rights are restored to the unprivileged domain.

An implicit requirement of MPKAlloc is that the allocator must place metadata and user data on separate memory pages. So, while glibc's ptmalloc is an important allocator to protect, its design decision to place metadata inline with user memory chunks precludes straightforward use of MPK for mitigation. However, several other popular allocators do in fact satisfy this requirement, including tcmalloc and PartitionAlloc, which are both used by Chromium and, thus, are heavily targeted by attackers. tcmalloc tracks spans, or user allocations at page granularity, using a trie held by `MetaDataAllocator` objects located on separate memory pages. PartitionAlloc instead uses a super page structure for similar purposes that is also stored in dedicated memory pages. Thus, critical metadata that is targeted by Chrome heap exploits can easily be protected by MPK and MPKAlloc by placing that metadata on pages labeled with a distinct protection key.

ThreadLock [3] similarly builds on MPK to efficiently enforce principal isolation at the thread level.

Neural Program Analysis

The final research area my group is currently focusing on exploring neural program analyses for security. As a first step, we have examined the efficacy and limitations of deep neural networks as applied to fundamental binary analysis tasks such as disassembly and function boundary recognition [4]. In this work, we demonstrate a number of concrete techniques for introducing unacceptably high misclassification rates in two recent examples of neural disassembly and function boundary recognition models. Our evaluation demonstrates that attackers would be able to easily evade defensive analysis pipelines built on these models. We conclude that a straightforward application of even state-of-the-art model architectures, e.g., BERT-based Transformers, ignores crucial domain-specific semantic information that in turn will leave the door open for attackers to elude detection by such models. Using insights from this work, my group is currently developing novel model embeddings and architectures that close this gap in order to realize the scalability and powerful inference capabilities that neural techniques promise to bring to program analysis.

Bibliography

[1] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. In *Proceedings of the ISOC Network and Distributed System Security Symposium*, February 2020. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2020.24415.

[2] William Blair, William Robertson, and Manuel Egele. MPKAlloc: Efficient Heap Meta-data Integrity Through Hardware Memory Protection Keys. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2022. Springer International Publishing, Cham, 136–155. https://doi.org/10.1007/978-3-031-09484-2_8.

[3] William Blair, William Robertson, and Manuel Egele. ThreadLock: Native Principal Isolation Through Memory Protection Keys. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, July 2023. ACM, Melbourne VIC Australia, 966–979. https://doi.org/10.1145/3579856.3595797.

[4] Joshua Bundt, Michael Davinroy, Ioannis Agadakos, Alina Oprea, and William Robertson. Black-Box Attacks Against Neural Binary Function Detection. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*, September 2023.

[5] Joshua Bundt, Andrew Fasano, Brendan Dolan-Gavitt, William Robertson, and Tim Leek. Evaluating Synthetic Bugs. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, 2021. 14–15.

[6] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Timothy Leek, Andrea Mambretti, William Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: Large-Scale Automated Vulnerability Addition. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2016. 110–121. https://doi.org/10.1109/SP.2016.15.

[7] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, Davide Balzarotti, and William Robertson. SoK: Enabling Security Analyses of Embedded Systems via Rehosting. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, May 2021. ACM, Virtual Event Hong Kong, 687–701. https://doi.org/10.1145/3433210.3453093.

[8] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. TriggerScope: Towards Detecting Logic Bombs in Android Applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2016. 377–396. https://doi.org/10.1109/SP.2016.30.

[9] A. Kharraz, W. Robertson, and E. Kirda. Surveylance: Automatically Detecting Online Survey Scams. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2018. 70–86. https://doi.org/10.1109/SP.2018.00044.

[10] Amin Kharraz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware. In *Proceedings of the USENIX Security Symposium*, 2016. 757–772. Retrieved from https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kharaz.

[11] William Koch, Abdelberi Chaabane, Manuel Egele, William Robertson, and Engin Kirda. Semi-Automated Discovery of Server-based Information Oversharing Vulnerabilities in Android Applications. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017. ACM, New York, NY, USA, 147–157. https://doi.org/10.1145/3092703.3092708.

[12] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *Proceedings of the ISOC Network and Distributed System Security Symposium*, 2017. https://doi.org/10.14722/ndss.2017.23414.

[13] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Speculator: A Tool to Analyze Speculative Execution Attacks and Mitigations. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC '19)*, December 2019. Association for Computing Machinery, San Juan, Puerto Rico, 747–761. https://doi.org/10.1145/3359789.3359837.

[14] Andrea Mambretti, Kaan Onarlioglu, Collin Mulliner, William Robertson, Engin Kirda, Federico Maggi, and Stefano Zanero. Trellis: Privilege Separation for Multi-user Applications Made Easy. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (Lecture Notes in Computer Science)*, 2016. Springer International Publishing, 437–456.

[15] Seyed Ali Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. Cached and Confused: Web Cache Deception in the Wild. In *Proceedings of the USENIX Security Symposium*, August 2020.

[16] Collin Mulliner, William Robertson, and Engin Kirda. Hidden GEMs: Automated Discovery of Access Control Vulnerabilities in Graphical User Interfaces. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2014. 149–162. https://doi.org/10.1109/SP.2014.17.

[17] Kaan Onarlioglu, Collin Mulliner, William Robertson, and Engin Kirda. PrivExec: Private Execution as an Operating System Service. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2013. 206–220. https://doi.org/10.1109/SP.2013.24.

[18] Kaan Onarlioglu, William Robertson, and Engin Kirda. Eraser: Your Data Won't Be Back. In *Proceedings of the IEEE European Symposium on Security and Privacy*, April 2018. 153–166. https://doi.org/10.1109/EuroSP.2018.00019.

[19] Michael Weissbacher, Tobias Lauinger, and William Robertson. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In *Research in Attacks, Intrusions and Defenses (Lecture Notes in Computer Science)*, 2014. Springer International Publishing, 212–233.

[20] Michael Weissbacher, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. ZigZag: Automatically Hardening Web Applications Against Client-side Validation Vulnerabilities. In *Proceedings of the USENIX Security Symposium*, 2015. USENIX Association, Berkeley, CA, USA, 737–752. Retrieved from http://dl.acm.org/citation.cfm?id=2831143.2831190.