Vasileios Kemerlis — *Research Statement*

My research focuses on building secure, trustworthy systems. Software systems impact virtually every aspect of modern society, ranging from economy and politics to science and education. Today's software systems, however, are large, complex, and plagued with vulnerabilities that allow perpetrators to exploit them for profit. The constant rise in the number of software weaknesses, coupled with the sophistication of modern cyber-adversaries, make the need for *effective*, *adaptive*, and *targeted* protection more critical than ever; even more so when it comes to *shielding critical infrastructure.* My work aims to improve the security posture of software systems, through a synergy of innovative defenses and exploit prevention techniques.

My research approach is characterized by the interplay between offense and defense. Both are equally important: attacking systems allows us to understand *how* software is insecure and *why* deployed defenses are failing, so that we can later design robust countermeasures. I first applied this principle throughout my dissertation work on operating systems security, introducing **ret2dir** [28]: a novel kernel exploitation technique that unveiled how standard OS abstractions, and conventional design practices, significantly weaken the effectiveness of state-of-the-art kernel protection mechanisms. More recently, I also proposed **EPF** [4]: another set of innovative kernel exploitation techniques, which rely on abusing the (e)BPF infrastructure, for bypassing memory safety protections that revolve around strong kernel-userland separation. In a similar vein, I co-invented the first cache-based side-channel attack that can be entirely executed in JavaScript context ("**Spy in the Sandbox**") [12], demonstrating that microarchitectural attacks are a realistic threat for modern web platforms. Lastly, my group and I exposed the risks to safety and privacy from the use of ROS (Robot Operating System; *i.e.,* the most widely-used robotics middleware), divulging how to leak image sensor information from, and actuate, physical robots in a completely unauthorized manner (**ROSSec**) [7].

On the defense side, I always seek practical solutions that have direct impact on real-world problems, usually in the form of *robust* systems and tools that can be readily *deployed* and *measured*. To this end, I designed and built **BPF-{ISR, NX, CFI}** [4], **xMP** [5], **kR^X** [9, 24], as well as **XPFO** [28] and **kGuard** [31, 34] (the latter two as part of my dissertation research), to harden contemporary OSes against attacks that exploit memory safety vulnerabilities in kernel code To protect generic C/C++ software against a wide range of threats, including code-reuse-based control-flow hijacking attacks, I co-developed **CCR** [8] and **Shuffler** [10]. The former enables the fast and robust fine-grained code randomization on end-user systems (post-compilation), by augmenting binaries with transformation-assisting metadata, while the latter continuously re-randomizes the code of a running program, including itself, thwarting code-reuse attacks by rapidly obsoleting leaked code layouts. As a professor at Brown University, I have also organized, and led, multiple security-related projects in collaboration with researchers and students from both industry and academia: **IvySyn** [3], **Egalito** [6], **RETracer** [11], **SysXCHG** [2]/**sysfilter** [16], **Nibbler** [17, 23], **Quack** [1], **BinWrap** [14], **VTPin** [18], **DynaGuard** [19], **μSCOPE** [15], **NaClDroid** [20], **EPI** [21]/**Polyglot** [22], and **FineIBT** [13], are all examples of such projects, related to fuzz testing, binary rewriting, crash triaging, sandboxing, debloating and hardening, compartmentalization, Android security, and hardware security, which I will describe later in this statement.

In the past, during my years as a Ph.D. candidate, I also focused on hardening binary-only applications against information leaks. Specifically, I built **ShadowReplica** [29], **TFA** [32], and **libdft** [33]: a set of tools that dynamically and transparently retrofit information-flow tracking (IFT) capabilities in multiprocess and multithreaded programs, outperforming previous solutions in many aspects. Lastly, I co-developed **VAP** [30] to thwart memory corruption attacks by fabricating unpredictable environments for adversaries; VAP dynamically partitions software and adapts (at run time) the defenses deployed, based on the executing part of a program, creating a "moving target" for attackers. Next is a summary of the real-world impact of my work, followed by my current and past research thrusts, and my vision for future research.

▶ **Impact and Technology Transfer.** Following its publication, RETracer has been adopted by Microsoft (*i.e.,* made part of the Windows Error Reporting platform) as the primary solution for triaging crashes, while Apple, Mozilla, and Tor, hardened their JavaScript engine(s) in light of our findings regarding cache-based side-channel attacks ("Spy in the Sandbox"). In addition, Egalito, Nibbler, and sysfilter, were all selected

by DoD/ONR to further explore for potential transition to practice (TTP) opportunities, while sysfilter has also been adopted by Star Lab (a subsidiary of Wind River) as part of their Linux-hardening solution (*i.e.,* Kevlar Embedded Security). More importantly, IvySyn has helped the TensorFlow and PyTorch framework developers to identify and fix 61 previously-unknown security vulnerabilities, and was awarded with 39 unique CVEs. Lastly, FineIBT has been adopted by Intel, and (a variant of it) was more recently upstreamed to the Linux kernel, and BinWrap won the Distinguished Paper Award at ACM ASIA CCS 2023.

In a similar vein, Shuffler was featured in ACM TechNews, CCR was a finalist (top 10) in the Applied Research competition, at the Cyber Security Awareness Week (CSAW) 2018, and ROSSec received articles in WIRED and in Kaspersky's Securelist. kGuard received recognition from both industry and academia—it was featured as the lead (invited) article in `;login:`, the technical journal of USENIX, and was one of the finalists for AT&T's Applied Research Paper award, at CSAW 2012. Moreover, after the publication of ret2dir, I presented my methodology for bypassing hardware-based kernel protections at Black Hat EU 2014, the premier industry conference on offensive security—this lead to ret2dir being discussed extensively in press and social media, receiving articles in LWN.net, the Linux Journal, and Dark Reading, and notable mention in Hacker News/Reddit. In addition, Qualcomm used parts of the code I released to perform (ret2dir-related) security assessments on its Linux-MSM kernel, while OpenBSD developers introduced kernel changes solely to mitigate ret2dir attacks. On top of that, the project was awarded with the first place in the Applied Research competition, at CSAW 2014, and was nominated for a Pwnie award in 2015 ("Most Innovative Research" category). Committed to real-world impact, we showcased kR^X at Black Hat USA 2017.

Following the publication of libdft, the research community endorsed the project and further improved my prototype in ways beyond its intended purpose. Indicatively, libdft has been used for: run-time error repair and containment (RCV project; MIT); data de-obfuscation, high-fidelity data provenance, custom memory allocator detection, evolutionary fuzzing, and automated exploitation (projects Carter, DataTracker, MemBrush, MAPScanner, VUzzer, TIFF, and Newton; VU University Amsterdam); as well as malware dissection, forensic investigation, system-wide IFT, and context-sensitive CFI. In addition, many studies use libdft nowadays as a standard benchmark in terms of IFT performance and/or effectiveness, while my original research prototype has been used for plagiarism detection, as well as the substrate for a new security architecture by BMW that safely allows the use of third-party applications in automotive settings. In addition, ShadowReplica, our follow-up work on libdft, was awarded with the second prize in the "CyberSecurity for the Next Generation" 2014 conference, organized by Kaspersky Labs, while "Practical Binary Analysis", a recent book on reverse engineering [26], is using libdft as a teaching apparatus. Due to the reasons above, libdft was the finalist (top 5) for the Artifacts Competition and Impact Award at ACSAC 2022 [25].

Operating Systems Security

The security of a computer system can only be as good as that of the underlying Operating System (OS) kernel. By compromising the kernel, attackers can escape isolation and confinement mechanisms, bypass access control and policy enforcement, and elevate privileges. As part of my work at Brown University, I developed novel protection mechanisms and introduced new concepts and techniques to secure *commodity* OSes against attacks that exploit memory safety vulnerabilities in kernel code. The following is a summary of four projects, namely **EPF** [4], **xMP** [5], **kR^X** [9, 24], and **μSCOPE** [15], which represent my recent contributions toward increasing the *trustworthiness* of major OSes, such as Linux, whose security posture affects millions of machines, ranging from mobile devices to high-end cloud servers. (OS kernel security was also the topic of my dissertation, which I describe extensively in the extended version of my research statement [27].)

► **kR^X [9, 24].** At the outset of my tenure as a professor at Brown, I focused on emerging OS kernel threats. Mirroring the co-evolution of attacks and defenses in user space, kernel exploits started to rely heavily on *code-reuse* techniques, like {return, jump, call}-oriented programming (ROP/JOP/COP), wherein attackers "compile" their shellcode by stitching together code fragments (*i.e.,* gadgets) from the executable sections of the kernel. To provide a *comprehensive* solution to this overarching problem, I conceived, launched, and led kR^X: an ambitious project aiming at developing new protection mechanisms, and exploit prevention

techniques, for combating a wide range of code-reuse threats against OS kernels, including ROP/JOP/COP and the particularly pernicious JIT-ROP. (The latter abuses *memory disclosure* vulnerabilities to pinpoint the location of gadgets, at run time, and assemble functional code-reuse payloads on-the-fly.) kR^X builds upon two main pillars: *(i)* the R⊕X memory policy, and *(ii)* fine-grained KASLR.

With the former, I argue about a new *safety property*, whereby kernel memory should be either readable or executable. The latter constitutes the cornerstone of kR^X: *i.e.,* a set of extensive *code randomization* techniques, specifically tailored to the kernel setting. The gist of kR^X lies on the interplay between: *(a.)* R⊕X, which ensures the secrecy of kernel code and prevents JIT-ROP, as the exact location of gadgets can no longer be disclosed at run time; and *(b.)* fine-grained KASLR, which thwarts ROP/JOP/COP and alike attacks, by perturbing kernel code, at the function and basic block level, so that the location of gadgets becomes unpredictable—code-reuse exploits rely on gadgets located at pre-determined memory addresses.

The distinctive aspect of kR^X is its *self-protection* approach to R⊕X, inspired by SFI (Software Fault Isolation) and similar techniques. Aiming to explore a wide spectrum of settings and trade-offs, my group and I designed and developed: `kR^X-SFI`, a software-only R⊕X scheme; and `kR^X-MPX`, a hardware-assisted R⊕X scheme, which leverages Intel MPX (Memory Protection Extensions) to minimize the protection overhead. Notably, kR^X was one of the systems that pioneered the use of MPX for SFI-like confinement. `kR^X-{SFI, MPX}` target modern, 64-bit platforms, instrumenting (at compile time) memory loads with *range checks* (RCs), which ensure (at run time) that the respective memory read operations do not leak code memory, effectively implementing an *execute-only memory* (XOM) scheme for contemporary architectures that lack native support for XOM. Lastly, through a set of compiler optimizations, fine-tuned for RCs operating on non-canonical memory layouts (*e.g.,* x86-64), we demonstrated that the comprehensive and low-overhead enforcement of R⊕X is feasible, without a hypervisor or a super-privileged hardware component.

▶ `xMP` [5] & μ`SCOPE` [15]. In anticipation that code-reuse attacks will progressively extenuate, due to defense schemes like kR^X (XOM plus code diversification), or CFI (Control-Flow Integrity), my focus shifted to the design of principled protection mechanisms against *data-oriented* attacks: *i.e.,* attacks that exploit memory safety vulnerabilities to corrupt, or leak, critical and sensitive data without hijacking the control flow. To this end, I conceived and co-lead, along with colleagues from the Technical University of Munich and Stony Brook University, xMP: an aspiring project aiming at developing new OS abstractions and mechanisms for enabling OS kernels and userland applications to protect critical/sensitive data against data-oriented attacks.

With xMP, we introduced two innovative hardening techniques: *(i)* selective memory protection, and *(ii)* context-bound pointer integrity. The former allows assigning memory to multiple, different *protection domains*, and *dynamically switching* between such domains based on *execution context*. Protection domains enforce different access policies (*i.e.,* memory can be {read, write, ...}-protected), while an execution context can be any code snippet, ranging from whole functions to single instructions, effectively enabling *adaptive* protection: *e.g.,* sensitive data can be mapped as non-accessible in, say, `domain[0]` and readable, or `RW`, in `domain[1]`, with the application/kernel executing in `domain[0]` for the most part and selectively (and temporarily) switching to `domain[1]` only to access, or update, the sensitive data. Context-bound pointer integrity provides *referential integrity* to data that are selectively-protected, preventing attacks that rely on tampering-with pointers to such data (*e.g.,* the attacker corrupts a pointer to a protected data structure to make it point to a different protected data structure, or a counterfeit/injected one); it also guarantees that pointers to selectively-protected data can be dereferenced, or updated, only in the right execution context.

We implemented *(i)* by taking advantage of virtualization extensions available in commodity CPUs, such as EPTP switching (`VMFUNC`) and virtualization exceptions (`#VE`), and *(ii)* by leveraging an HMAC-like scheme that we specially-crafted for short input values, like memory addresses, demonstrating comprehensive protection for a plethora of diverse data structures, such as page tables and process credentials in the Linux kernel, and cryptographic material in userland applications (*e.g.,* OpenSSL, OpenSSH). It is also worth noting that Intel has recently announced an extension, dubbed MPK/PKU (Memory Protection Keys/Protection Keys for Userspace), for enabling selective memory protection. MPK is only available in high-end Skylake-SP Xeon CPUs and supports 16 protection domains; xMP, on the other hand, builds upon salient, widespread CPU features and can support up to 512 protection domains.

More recently, along with a team of colleagues from Rice, Penn, UIUC, and EPFL, I co-developed a new methodology for automatically identifying *(a.)* compartmentalization and *(b.)* privilege-reduction opportunities in large, monolithic codebases, such as the Linux kernel. μSCOPE instruments and profiles software activity at the granularity of instructions and objects, encoding each reference (*i.e.,* privilege requirement) in a novel low-level access control matrix, the CAPMAP (Context-Aware Privilege Memory Access Pattern). μSCOPE then uses CAPMAP as the ground truth with which it compares competing software *compartmentalization hypotheses*. As part of this work, we also introduced a set of metrics that allow μSCOPE to evaluate the level of privilege separation that is possible for a given compartmentalization strategy, compared to both monolithic (*i.e.,* fully over-privileged) and the minimum-required-to-run (*i.e.,* least-privilege) baselines, and estimate the cost of enforcement for a range of potential isolation mechanisms, including xMP.

▶ **`EPF` [4].** With defenses like kR^X and xMP in place, which thwart kernel control-flow hijacking and data-oriented attacks, I recently started investigating the security ramifications of allowing userland applications *delegate* computations to the OS kernel, via mechanisms like BPF (Berkeley Packet Filter). BPF empowers user space processes with the ability to offload the execution of certain tasks to kernel space, or further customize kernel functionality, by effectively acting a "universal" in-kernel virtual machine. To this end, I conceived, launched, and led EPF (Evil Packet Filter): an offensive project for studying the impact of BPF on the security and robustness of the OS kernel itself. Specifically, I introduced two novel attacks, dubbed EPF-v1 (*BPF-Reuse*) and EPF-v2 (*BPF-ROP*), which demonstrate how (e)BPF can be abused to *facilitate* both code-reuse- and data-oriented-based exploits, and bypass deployed (kernel) defenses for such threats.

EPF-v1/BPF-Reuse leverages memory corruption vulnerabilities (in kernel code) to cause the BPF subsystem execute a benign/verified (e)BPF program from a specific *offset* (*i.e.,* skip a couple of bytes in the respective instruction stream). This effectively results in executing *unverified* (e)BPF instructions, and hence bypassing all the constraints imposed by the BPF runtime, escape the BPF sandbox, and further facilitate attacks that evade access control and policy enforcement, and result in privilege escalation. (EPF-v1 "hides" *malicious BPF code* inside benign BPF programs that *cannot* be rejected by the BPF verifier.) EPF-v2/BPF-ROP uses valid/verified (e)BPF code to encode code-reuse (*e.g.,* ROP) payloads, thereby facilitating the unconstrained *injection* of such payloads in kernel space, which is a necessary step for mounting code-reuse-based attacks. (EPF-v2 "hides" *gadget addresses* inside benign BPF programs.)

As part of this work, I also designed three innovative, low-overhead defenses, namely BPF-{ISR, NX, CFI}, to harden the BPF infrastructure against EPF-style attacks. BPF-ISR builds upon the concept of instruction-set randomization, and thwarts EPF-v2 by *randomizing* the in-kernel representation of (e)BPF code, hence preventing an attacker from reliably using (e)BPF instructions to encode gadget addresses. BPF-NX defines an integrity-protected *sub-address space* for (e)BPF programs, and along with BPF-CFI that hinders the execution of (e)BPF programs from *random offsets*, à la CFI, they mitigate EPF-v1 by ensuring that only verified (e)BPF instructions are executed at run time.

Systematic Software Hardening

Shortly after joining Brown, I embarked on the challenging task of developing *generic* and *robust* mechanisms for armoring C/C++ software against a plethora of attacks that recon on memory corruption vulnerabilities. **`SysXCHG`** [2], **`Egalito`** [6], **`CCR`** [8], **`Shuffler`** [10], **`sysfilter`** [16], and **`Nibbler`** [17], which I briefly describe in what follows, are six projects that represent my main contributions to this research strand. I have also proposed solutions for various emerging threats: *e.g.,* to harden PHP code against attacks that abuse deserialization vulnerabilities, I co-designed **`Quack`** [1]—a static analysis framework that automatically restricts the set of classes allowed at deserialization points/sites, via means of novel *duck-typing inference*, effectively limiting program-code reuse through object injection (*i.e.,* property-oriented programming); to protect C++ programs from **`vtable`** *hijacking*, a prevalent C++ exploitation technique, I co-designed **`VTPin`** [18]; to defend applications from *canary brute-force* attacks, I designed and co-developed **`DynaGuard`** [19]; to *sandbox* native code/modules in (1) Android Apps, I built **`NaClDroid`** [20], and (2) Node.js Apps, I built **`BinWrap`** [14]. (✹ The latter won the Distinguished Paper Award at ACM ASIA CCS 2023.)

▶ **`Shuffler` [10], `Nibbler` [17, 23], `sysfilter` [16], `SysXCHG` [2].** The universal adoption of NX (Non-eXecutable memory) by contemporary platforms promoted code-reuse as the standard technique for exploiting memory corruption vulnerabilities in userland applications. While ASLR (Address Space Layout Randomization), or code randomization in general, breaks "canned" code-reuse payloads that rely on gadgets placed at fixed memory locations, memory disclosure vulnerabilities are usually employed by JIT-ROP-based exploits to bypass the diversification. Building upon our earlier ideas on code diversification in kernel settings (kR^X, kGuard), along with students and colleagues from Columbia University and University of British Columbia, we designed and built Shuffler: the first egalitarian, live code re-randomization framework. Shuffler takes the idea of code diversification to the extreme, by *continuously* re-randomizing executable code, at run time, in the order of milliseconds, introducing a hard deadline on JIT-ROP attacks. To this end, we developed two innovative mechanisms: *(i)* code pointer abstraction, and *(ii)* asynchronous migration of program execution. The former *virtualizes* code pointers, enabling Shuffler to relocate code (at run time) without tracking code-pointer values. The second *expedites* code re-randomization, as it allows the rapid migration of thread execution, from one (randomized) copy of the code to the next, without stop-the-world pauses. The distinctive characteristic of Shuffler, however, is *egalitarianism—i.e.,* Shuffler can shuffle itself!

In addition, my group and I, in collaboration with students and colleagues from Columbia University and Stevens Institute of Technology, built Nibbler: a *software debloating* framework for removing unused code from binary shared libraries, producing "thin variants" that contain only the absolute necessary code for supporting the application(s) they link-with. Nibbler employs novel static binary analysis techniques for *safely* removing unused library code (*i.e.,* code that never executes, under any given input), further boosting defenses like Shuffler (*e.g.,* ≈20% faster code re-randomization). More recently, we further extended Nibbler's program analyses to statically extract system call (syscall) sets from binary-only applications in a scalable, precise, and complete (safe over-approximation) manner. sysfilter, our resulting framework, automatically *(a.) limits what OS services attackers can (ab)use*, by enforcing the principle of least privilege (PoLP) with respect to the system call API, and *(b.) reduces the attack surface of the kernel*, by restricting the system call set available to userland processes. Moreover, we used sysfilter to perform the largest (to date) study regarding system call usage, analyzing ≈30K applications (*i.e.,* the complete set of C/C++ binaries in Debian), reporting interesting insights about syscall set sizes, most- and least-frequently used syscalls, syscall site distribution (libraries vs. main binary), and more. Lastly, to allow programs that invoke other programs (*e.g.,* via `execve[at]`) execute in accordance to the PoLP, I designed SysXCHG: an innovative least-privilege syscall-filtering enforcement mechanism that addresses the limitations of `seccomp-BPF`. Specifically, in contrast to the current, hierarchical design of `seccomp-BPF`, which does not permit a program to run with a different set of allowed syscalls than its descendants, SysXCHG enables applications to run with "tight" syscall filters, uninfluenced by any future-executed (sub-)programs, by allowing filters to be *dynamically exchanged* at runtime (safely) during `execve[at]`. As a part of SysXCHG, we also introduced `xfilter`: a new mechanism for fast syscall filtering, which relies on *process-specific views* of the kernel's syscall table. (✹ Nibbler and sysfilter were selected by DoD/ONR to further explore for potential transition to practice opportunities. sysfilter has also been adopted by Star Lab—*i.e.,* a subsidiary of Wind River—as part of their Linux-hardening solution, namely Kevlar Embedded Security.)

▶ **`CCR` [8] & `Egalito` [6].** At the time of its publication, Shuffler achieved the *fastest* code re-randomization rate(s), with moderate overhead. Motivated by this outcome, along with students and colleagues from Stony Brook University and Northeastern University, we explored the idea of promoting code diversification to a first-rate principle. Specifically, we designed and developed CCR: a compiler-rewriter framework, accompanied by a rigorous methodology, which enables the *fast* and *robust* code randomization on end-user systems. Built atop LLVM/Clang, CCR collects and embeds *transformation-assisting metadata* into compiled binaries, which allow the generation of hardened variants, post-compilation, without relying on code disassembly or other ad-hoc program analyses. Our evaluation results indicate that fine-grained code randomization can be performed at the client side, during installation or at load time, with *no* run-time overhead. Lastly, CCR allows reversing code transformations to maintain compatibility with debugging, code signing, *etc.*

More recently, I revisited the assumption of requiring compiler-emitted metadata to perform post-compilation code transformations in a precise and safe manner. Specifically, motivated by recent advances in modern compiler toolchains, such as (1) the strict separation between code and data (in the output binaries), (2) the emission of position-independent code by default (to enable full ASLR), and (3) the inclusion of exception handling metadata in all binaries (to facilitate interoperability with C++), I conceived and co-led, along with colleagues from Columbia University, Egalito: the first *layout-agnostic binary recompilation framework*. Egalito leverages (1)–(3) to perform a set of novel and principled analyses that allow transforming stripped binary code, without requiring patching or virtualization, in a *complete* and *safe* manner. Egalito supports rewriting binaries *in-place*, by lifting binary code into a *layout-agnostic*, machine-specific intermediate representation (IR), dubbed EIR, and then allowing "tools"—structured as (re)compiler passes—to inspect or alter it. To date, we have used Egalito to implement a wide range of such tools, including live code re-randomization and debloating runtime environments (in the spirit of Shuffler and Nibbler), profile-guided code optimizers, hardening frameworks (for retrofitting protections like retpolines and shadow stacks to binary-only C/C++ software), as well as utilities for speeding up (by 15x–60x) popular fuzzers, like AFL. (✹ Egalito was also selected by DoD/ONR to further explore for potential TTP opportunities.)

Other Research

During my time at Brown, I had the opportunity to collaborate with researchers, and supervise students, on a wide variety of security-related topics, including fuzz testing, crash triaging, microarchitectural attacks, as well as hardware and robotics security. The following is a summary of the most recent of these efforts.

Fuzz Testing. Deep learning (DL) frameworks, such as PyTorch and TensorFlow, are increasingly used in *safety-critical* settings, like autonomous driving, providing a set of rich high-level APIs to model developers, in order to allow them integrate DL functionality into end-to-end, real-world systems. These developer-accessible APIs perform DL-specific operations, and are usually implemented in managed languages, like Python. Yet, the essential parts of such operations—which are called *kernels*—are implemented in *memory-unsafe* languages, like C/C++, to boost performance and to allow for execution in different platforms (*e.g.,* CPUs, GPUs, TPUs). To automatically discover memory safety vulnerabilities in DL frameworks, I designed **`IvySyn`** [3]: the first fully-automated system for uncovering memory errors in DL frameworks. IvySyn leverages the statically-typed nature of native APIs in order to automatically perform *type-aware* mutation-based fuzzing on low-level kernel code. Given a set of offending inputs that trigger memory safety (and runtime) errors, IvySyn automatically *synthesizes* code snippets in high-level languages (*e.g.,* in Python), which propagate error-triggering input via high(er)-level APIs. Such code snippets essentially act as *"Proof of Vulnerability"*, as they demonstrate the existence of bugs in native code that an attacker can target through various high-level APIs. (✹ IvySyn has already helped the TensorFlow and PyTorch framework developers to identify and fix 61 previously-unknown security vulnerabilities, and was awarded with 39 unique CVEs.)

Crash Triaging. Crash reporting is typically employed by large software providers (Microsoft, Google, Apple) to collect crash dumps and file bug reports against the right developer(s). One of the most critical aspects of this process is *crash triaging*: the batching of crashes that are *likely* caused by the same bug. However, the precise and accurate triaging of crashes is a notoriously hard problem, limiting deployed services to merely using stack traces for pinpointing bugs. To address the limitations of the current status quo, in collaboration with researchers from Microsoft Research, we designed and developed **`RETracer`** [11]: a system for triaging crashes based on program semantics reconstructed from *partial* memory dumps. RETracer builds upon two key techniques that mimic how developers usually analyze crash reports: *(i)* backward data flow tracking, and *(ii)* reverse program execution. In tandem, they facilitate the investigation of how bad memory values (*e.g.,* a corrupted pointer) contribute to a crash, thereby aiding *root cause analysis*. (✹ RETracer has been adopted by Microsoft as the primary solution for crash triaging, processing millions of crash reports per day and outperforming the previous triaging system of WER—*i.e.,* Windows Error Reporting—by eliminating more than two-thirds of the triage errors.)

Microarchitectural Attacks. Motivated by the sheer number of microarchitectural attacks against modern computer systems, along with colleagues from Columbia University, we investigated the feasibility of side-channel threats in contemporary web platforms. However, what began as a quest to quantify *information leakage* in browser settings, resulted in the *first* cache-based side-channel attack, which we named "**`Spy in the Sandbox`**" [12], that can be entirely executed in JavaScript (JS) context (*i.e.,* without running native code on the victim's machine). Specifically, we proposed a set of techniques for: *(i)* tracking browsing activity, even when the "private browsing" mode is used; *(ii)* constructing covert channels inside the JS sandbox; and *(iii)* detecting certain hardware events (*e.g.,* mouse and network activity, sensor interrupts). (✹ In light of our findings, Apple, Mozilla, and Tor, limited the time resolution of the JS performance API to counter JS-based microarchitectural attacks on user privacy.)

Hardware and Robotics Security. Once CFI (Control-Flow Integrity) started gaining traction in real-world settings (as a defense/solution against control-flow hijacking attacks), I begun investigating efficient and effective CFI policy enforcement mechanisms. More specifically, in collaboration with Intel Corporation, I designed and co-developed **`FineIBT`** [13]: a modern CFI *enforcement scheme* that improves the precision of hardware-assisted CFI solutions, like Intel IBT, by instrumenting program code to further reduce the valid/-allowed targets of indirect forward-edge transfers. I studied the design of FineIBT on the x86-64 architecture, and proposed compact instrumentation stubs that incur low runtime and memory overheads, which, however, are generic enough, so as to support different CFI policies. FineIBT incurs negligible runtime slowdowns and outperforms Clang-CFI, while providing complete coverage in the presence of modern software features. More importantly, it supports a wide range of CFI policies with the same, *predictable* performance. (✹ FineIBT has been adopted by Intel and, a variant of it, was recently upstreamed to the Linux kernel.)

Instruction-set randomization (ISR) has been proposed as an alternative to non-executable memory for countering *code-injection* attacks. In collaboration with Columbia University, we designed and developed **`Polyglot`** [22]: the *first* hardware-based ISR system that (naturally) protects against code injection attacks, but can also mitigate *code-reuse*, if combined with leakage-resilient code diversification (*e.g.,* fine-grained KASLR; the code randomization scheme we introduced with kR^X). Polyglot, in antithesis to earlier ISR prototypes, *(a.)* utilizes strong encryption (*i.e.,* AES and ECC), *(b.)* supports code sharing, and *(c.)* is applicable to the entire software stack (*i.e.,* bootloader, hypervisor, OS kernel, and user applications). We also introduced **`EPI`** [21]: an efficient *tag-based pointer integrity* companion mechanism that is tailored to microcontrollers and embedded devices. Lastly, along with students and colleagues from Brown, we designed and launched **`ROSSec`** [7]: the first large scale scan of the entire IPv4 Internet address space for *exposed instances* of the Robot Operating System (ROS)—*i.e.,* the most widely-used robotics research platform. We identified numerous publicly-accessible ROS hosts that allowed access, in an *unauthorized manner*, to robotic sensors and actuators, and demonstrated (with permission) the risks to user safety and privacy by leaking image sensor information from, and actuating, physical robots present at major US universities.

Future Directions

As commodity software is increasingly used in mission-critical settings, improving the trustworthiness of key components in the software stack becomes an issue of utmost importance. To this end, I introduced techniques to protect essential elements, like OS kernels (projects EPF [4], xMP [5], kR^X [9, 24], XPFO [28], kGuard [31, 34]), trusted applications, such as web servers, RDBMSes, and language runtime environments, written in memory-unsafe languages (projects Egalito [6], CCR [8], Shuffler [10], Quack [1], BinWrap [14], FineIBT [13]), and binary-only software (projects SysXCHG [2], sysfilter [16], Nibbler [17, 23], Shadow-Replica [29], TFA [32], libdft [33], VAP [30], VTPin [18]), against a range of attacks that exploit memory safety vulnerabilities. Yet, the bulk of the work lies ahead. Going forward, I plan to capitalize on my expertise and address the security challenges of the unique software ecosystems that emerging technologies, like cloud computing, Internet of things (IoT), and cyber-physical systems (CPS), have started shaping. The following research directions capture my vision towards *next-generation* software protection.

① **Agile Hardening.** Despite years of research, software hardening still revolves around the concept of defense in depth. The basic premise behind this strategy to protection is that by stacking together different exploit mitigation techniques, software compromise becomes harder. Yet, this "blanket" approach to software security (*i.e.,* "protect everything, the same way, all the time, at the same intensity") works well only with defences that have negligible, or close to zero, performance overhead and are oblivious to the setting(s) in which the hardened software is used. Hence, as security-critical software components are utilized verbatim in diverse domains—ranging from IoT to cloud computing environments—software vendors are coerced to employ "cheap" exploit mitigations with lax security guarantees, rather than protection primitives with assured security; the latter typically inflict non-negligible run-time slowdowns or operate on platforms with specific characteristics. Consequently, the majority of the software stack remains armored with rudimentary broad-spectrum defense schemes, such as ASLR, NX memory, and compiler-based tripwires (*e.g.,* stack canaries) despite the latest advances in the development of protections with guaranteed security properties.

Looking forward, I believe that we should rethink how we harden our systems. As essential software components move rapidly from one domain to another to meet the demands of emerging ecosystems (*e.g.,* app stores and microservice applications), which rely heavily on software of unknown quality and provenance, the problems of the blanket approach will only be exacerbated. The preclusion of protection mechanisms that do not align with the "always-on, applicable-everywhere" mode of operation culminates in a monoculture of defenses (*i.e.,* the same exploit mitigations are used in environments with fundamentally different security requirements and characteristics), allowing adversaries to automate their techniques for bypassing protections and reliably exploit software running in dissimilar settings with minimal (additional) effort.

To address these challenges, I envision a next-generation security architecture that enables defenses to be constantly in *flux*, fostering the deployment of robust protection mechanisms—heavyweight, yet principled and effective—as and where needed, strategically. My goal is to infuse systems with the ability to dynamically *adapt* their defenses along several dimensions, by elevating *hardening rectification* and *agility* to first-rate principles. The benefits of such an approach to software security are manifold. First, hardening agility creates a diversified and unpredictable environment, which naturally breaks the monoculture of defenses and hinders the ability of adversaries to use canned recipes for bypassing exploit mitigations. Second, hardening rectification allows software to make the best use of the hardening capabilities that a particular setting offers, and dynamically adapt the deployed defences to meet certain needs (*e.g.,* intensify hardening to protect against an active attack, or selectively turn off protections to increase performance/preserve power).

To support this effort, I plan to build upon our prior work on SysXCHG [2], Egalito [6], CCR [8], and VAP [30], and investigate techniques that efficiently, and transparently, enable systems to adapt the *(a.)* granularity (coarse-grain vs. fine-grain), *(b.)* intensity, and *(c.)* layering, of deployed protection mechanisms, based on the *(i)* environment in which the software is executing (*e.g.,* IoT or CPS setting, cloud platform), *(ii)* data that the application operates upon (high-value vs. low-value), *(iii)* innate properties of particular code parts (high-privileged vs. deprivileged/sandboxed), and *(iv)* external events. In addition, I plan to explore minimal *hardware extensions* for facilitating dynamic, fine-grained memory isolation and confinement—a prerequisite for adaptive protection—in a similar fashion to xMP [5] and FineIBT [13].

In the longer term, my aim is to leverage *automated program verification and testing* techniques for applying defense mechanisms strategically, in a precise and targeted manner: *e.g.,* only in code parts that certain safety properties cannot be proven to hold, or only when an application operates on inputs that cannot be proven to be safe, effectively coupling hardening with formal verification and fuzzing (à la IvySyn [3]). (✹ In 2021, I hired a postdoctoral researcher—Vaggelis Atlidakis, CIFellow 2020—to galvanize this effort, while more recently I won the NSF CAREER Award, securing $660K towards next-generation hardening.)

② **Kernel Security and Transient Execution Attacks.** Kernel protection is a fertile and challenging research area that I plan to continue working on, as kernel exploitation has currently turned into one of the primary methods for compromising hardened systems. ret2dir [28] and ret2usr [31, 34]—the two attacks I introduced as part of my dissertation work—signified the importance of proper isolation between the OS kernel and userland, by demonstrating how the weak separation of different protection domains can be

leveraged to subvert state-of-the-art defenses. Going forward, I plan to explore new OS abstractions, kernel designs, and hardware extensions, in the spirit of R⊕X [9, 24], xMP [5], and `BPF-{ISR, NX, CFI}` [4], for the robust and efficient *compartmentalization* of different protection domains. In the recent past, I secured $200K in funding from DoD/DARPA towards this goal. At the same time, my group and I are keep pushing the envelope on: *(1)* generic techniques regarding kernel exploitation and defense [37, 38]; and *(2)* novel protection mechanisms against emerging (OS) threats, such as transient execution attacks [35, 36].

③ **Diversification and Debloating.** Diversifying the internal structure of system components, coupled with the removal of unnecessary functionality, is an effective approach for breaking the assumptions of attackers regarding vulnerable targets, and challenging the construction of reliable exploits. Unfortunately, though several protections that rely on the concept of diversification and/or debloating have been proposed, only a few have actually seen widespread adoption. Based on our prior work on diversifying code [8, 9, 24, 10, 31], and my real-world experiences from implementing ASLR in Oracle Solaris, one of the main obstacles for the use of more extensive forms of artificial diversification and automated feature removal is that they clash with existing operations that rely on software uniformity (*e.g.,* debugging, crash reporting, code signing). I believe that software *diversity*, along with adaptive *debloating* [17, 23], will play a pivotal role in *comprehensive* system security, and I plan to investigate the application of these concepts in ways that will not hinder existing software analysis practices and verification mechanisms. In the recent past, I secured $1M in funding from DoD/ONR towards this goal, while I have also started investigating tactical hardware extensions for protecting software diversification [35, 13].

Lastly, as was the case with my past work, the research in the above areas will be driven by a set of guiding principles that I always seek to satisfy: minimal run-time overhead, transparent or oblivious operation, robustness against evasion attempts, and quantifiable results. Satisfying these requirements usually leads to the development of technologies and solutions that can be deployed on existing systems and have immediate impact. Seeking practical solutions to important, real-world security and privacy problems are exciting endeavors that I plan to continue pursuing along the above, as well as new, research areas.

References (post-2015, Brown)

✪: Top-tier venue, ___: Advisee

[1] ✪ Yaniv David, Neophytos Christou, Andreas D. Kellas, **Vasileios P. Kemerlis**, and Junfeng Yang. QUACK: Hindering Deserialization Attacks via Static Duck Typing. In *Proceedings of the 31st Network and Distributed System Security (NDSS) Symposium*, San Diego, CA, USA, February 2024. [**Acceptance rate: 17.6%**]

[2] ✪ Alexander J. Gaidis, Vaggelis Atlidakis, and **Vasileios P. Kemerlis**. SysXCHG: Refining Privilege with Adaptive System Call Filters. In *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*, Copenhagen, Denmark, November 2023.
[**Acceptance rate: 19.2%, Software artifact:** `https://gitlab.com/brown-ssl/sysxchg`]

[3] ✪ Neophytos Christou, Di Jin, Vaggelis Atlidakis, Baishakhi Ray, and **Vasileios P. Kemerlis**. IvySyn: Automated Vulnerability Discovery in Deep Learning Frameworks. In *Proceedings of the 32nd USENIX Security Symposium (USENIX SEC)*, pages 2383–2400, Anaheim, CA, USA, August 2023.
[**Acceptance rate: 30.1%, Software artifact:** `https://gitlab.com/brown-ssl/ivysyn`]

[4] ✪ Di Jin, Vaggelis Atlidakis, and **Vasileios P. Kemerlis**. EPF: Evil Packet Filter. In *Proceedings of the 29th USENIX Annual Technical Conference (USENIX ATC)*, pages 735–751, Boston, MA, USA, July 2023.
[**Acceptance rate: 18.4%, Software artifact:** `https://gitlab.com/brown-ssl/epf`]

[5] ✪ Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, **Vasileios P. Kemerlis**, and Michalis Polychronakis. xMP: Selective Memory Protection for Kernel and User Space. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*, pages 584–598, San Francisco, CA, USA, May 2020.
[**Acceptance rate: 12.3%, Software artifact:** `https://github.com/virtsec/xmp`]

[6] ✪ David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and **Vasileios P. Kemerlis**. Egalito: Layout-Agnostic Binary Recompilation. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 133–147, Lausanne, Switzerland, March 2020.
[**Acceptance rate: 18%, Software artifact:** `https://gitlab.com/egalito`]

[7] ✪ Nicholas DeMarinis, Stefanie Tellex, **Vasileios P. Kemerlis**, George Konidaris, and Rodrigo Fonseca. Scanning the Internet for ROS: A View of Security in Robotics Research. In *Proceedings of the 36th IEEE International Conference on Robotics and Automation (ICRA)*, pages 8514–8521, Montreal, Canada, May 2019.

[8] ✪ Hyungjoon Koo, Yaohui Chen, Long Lu, **Vasileios P. Kemerlis**, and Michalis Polychronakis. Compiler-assisted Code Randomization. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P)*, pages 472–488, San Francisco, CA, USA, May 2018.
[**Acceptance rate: 11.5%, Software artifact:** `https://github.com/kevinkoo001/CCR`]

[9] ✪ Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and **Vasileios P. Kemerlis**. kRˆX: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, pages 420–436, Belgrade, Serbia, April 2017.
[**Acceptance rate: 20%, Software artifact:** `https://gitlab.com/brown-ssl/krx`]

[10] ✪ David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, **Vasileios P. Kemerlis**, Junfeng Yang, and William Aiello. Shuffler: Fast and Deployable Continuous Code Re-randomization. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 367–382, Savannah, GA, USA, November 2016.
[**Acceptance rate: 18.1%**]

[11] ✪ Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and **Vasileios P. Kemerlis** RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 820–831, Austin, TX, USA, May 2016.
[**Acceptance rate: 19%**]

[12] ✪ Yossef Oren, **Vasileios P. Kemerlis**, Simha Sethumadhavan, and Angelos D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, pages 1406–1418, Denver, CO, USA, October 2015. [**Acceptance rate: 19.8%**]

[13] Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and **Vasileios P. Kemerlis**. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 527–546, Hong Kong, HK, October 2023. [**Acceptance rate: 23.5%, Software artifact:** `https://gitlab.com/brown-ssl/fineibt`]

[14] George Christou, Grigoris Ntousakis, Eric Lahtinen, Sotiris Ioannidis, **Vasileios P. Kemerlis**, and Nikos Vasilakis. BinWrap: Hybrid Protection against Native Node.js Add-ons. In *Proceedings of the 8th ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, pages 429–442 Melbourne, Australia, July 2023. [**Acceptance rate: 18.8%, Distinguished Paper Award**]

[15] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, **Vasileios P. Kemerlis**, Mathias Payer, Adam Bates, André DeHon, Jonathan M. Smith, and Nathan Dautenhahn. μSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 296–311, San Sebastian, Spain, October 2021. [**Acceptance rate: 23.9%, Software artifact:** `https://gitlab.com/fierce-lab/uscope`]

[16] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and **Vasileios P. Kemerlis**. sysfilter: Automated System Call Filtering for Commodity Software. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 459–474, San Sebastian, Spain, October 2020.
[**Acceptance rate: 25.6%, Software artifact:** `https://gitlab.com/brown-ssl/sysfilter`]

[17] Ioannis Agadakos, Di Jin, David Williams-King, **Vasileios P. Kemerlis**, and Georgios Portokalidis. Nibbler: Debloating Binary Shared Libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, San Juan, Puerto Rico, December 2019. [**Acceptance rate: 22.6%**]

[18] Pawel Sarbinowski, **Vasileios P. Kemerlis**, Cristiano Giuffrida, and Elias Athanasopoulos. VTPin: Practical VTable Hijacking Protection for Binaries. In *Proceedings of the 32nd Annual Computer Security Applications Conference (ACSAC)*, pages 448–459, Los Angeles, CA, USA, December 2016.
[**Acceptance rate: 22.8%, Software artifact:** `https://github.com/uberspot/vtpin`]

[19] Theofilos Petsios, **Vasileios P. Kemerlis**, Michalis Polychronakis, and Angelos D. Keromytis. DynaGuard: Armoring Canary-based Protections against Brute-force Attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, pages 351–360, Los Angeles, CA, USA, December 2015. [**Acceptance rate: 24.4%, Software artifact:** `https://github.com/nettrino/dynaguard`]

[20] Elias Athanasopoulos, **Vasileios P. Kemerlis**, Georgios Portokalidis, and Angelos D. Keromytis. NaClDroid: Native Code Isolation for Android Applications. In *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS)*, pages 422–439, Heraklion, Greece, September 2016. [**Acceptance rate: 21%**]

[21] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, **Vasileios P. Kemerlis**, and Simha Sethumadhavan. EPI: Efficient Pointer Integrity for Securing Embedded Systems. In *Proceedings of the 1st IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 163–175, Washington, DC, USA, September 2021.

[22] Kanad Sinha, **Vasileios P. Kemerlis**, and Simha Sethumadhavan. Reviving Instruction Set Randomization. In *Proceedings of the 11th IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 21–28, McLean, VA, USA, May 2017. [**Acceptance rate: 24.5%**]

[23] Ioannis Agadakos, Nicholas DeMarinis, Di Jin, Kent Williams-King, Jearson Alfajardo, Benjamin Shteinfeld, David Williams-King, **Vasileios P. Kemerlis**, and Georgios Portokalidis. Large-scale Debloating of Binary Shared Libraries. *ACM Digital Threats: Research and Practice (DTRAP)*, 1(4):1–28, December 2020. [**Software artifact:** `https://gitlab.com/brown-ssl/libfilter`]

[24] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and **Vasileios P. Kemerlis**. Kernel Protection against Just-In-Time Code Reuse. *ACM Transactions on Privacy and Security (TOPS)*, 22(1):5:1–5:28, January 2019.

[25] **Vasileios P. Kemerlis**. libdft: Dynamic Data Flow Tracking for the Masses. In *Annual Computer Security Applications Conference (ACSAC)*, Austin, TX, USA, December 2022.

[26] Dennis Andriesse. *Practical Binary Analysis*. No Starch Press, 2018.

[27] Vasileios P. Kemerlis. *Research Statement* (long version). `https://cs.brown.edu/~vpk/vpk-rs.long.pdf`

References (pre-2015, Columbia)

✪: Top-tier venue, ___: Mentee

[28] ✪ **Vasileios P. Kemerlis**, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking Kernel Isolation. In *Proceedings of the 23rd USENIX Security Symposium (SEC)*, pages 957–972, San Diego, CA, USA, August 2014. [**Acceptance rate: 19%, Software artifact:** `https://www.cs.columbia.edu/~vpk/research/ret2dir/`]

[29] ✪ Kangkook Jee, **Vasileios P. Kemerlis**, Angelos D. Keromytis, and Georgios Portokalidis. ShadowReplica: Efficient Parallelization of Dynamic Data Flow Tracking. In *Proceedings of the 20th ACM Computer and Communications Security Conference (CCS)*, pages 235–246, Berlin, Germany, October 2013. [**Acceptance rate: 19.8%**]

[30] ✪ Dimitris Geneiatakis, Georgios Portokalidis, **Vasileios P. Kemerlis**, and Angelos D. Keromytis. Adaptive Defenses for Commodity Software through Virtual Application Partitioning. In *Proceedings of the 19th ACM Computer and Communications Security Conference (CCS)*, pages 133–144, Raleigh, NC, USA, October 2012. [**Acceptance rate: 18.9%**]

[31] ✪ **Vasileios P. Kemerlis**, Georgios Portokalidis, and Angelos D. Keromytis. kGuard: Lightweight Kernel Protection against Return-to-user Attacks. In *Proceedings of the 21st USENIX Security Symposium (SEC)*, pages 459–474, Bellevue, WA, USA, August 2012. [**Acceptance rate: 19.4%, Software artifact:** `https://www.cs.columbia.edu/~vpk/research/kguard/`]

[32] ✪ Kangkook Jee, Georgios Portokalidis, **Vasileios P. Kemerlis**, Soumyadeep Ghosh, David I. August, and Angelos D. Keromytis. A General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware. In *Proceedings of the 19th Symposium on Network and Distributed System Security (NDSS)*, San Diego, CA, USA, February 2012. [**Acceptance rate: 18%**]

[33] **Vasileios P. Kemerlis**, Kangkook Jee, Georgios Portokalidis, and Angelos D. Keromytis. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, pages 121–132, London, UK, March 2012. [**Software artifact:** `https://gitlab.com/brown-ssl/libdft`]

[34] **Vasileios P. Kemerlis**, Georgios Portokalidis, Elias Athanasopoulos, and Angelos D. Keromytis. kGuard: Lightweight Kernel Protection. *USENIX ;login:*, 37(6):7–14, December 2012.

Work in Progress

___: Advisee

[35] Neophytos Christou, Alexander J. Gaidis, Vaggelis Atlidakis, and **Vasileios P. Kemerlis**. SMABlock: Preventing Speculative Memory-error Abuse with Artificial Data Dependencies. October 2023.

[36] Di Jin, Alexander J. Gaidis, and **Vasileios P. Kemerlis**. BPF-Box: Hardening BPF against Transient Execution Attacks. October 2023.

[37] Marius Momeu, Simon Schnückel, Kai Angnis, Michalis Polychronakis, and **Vasileios P. Kemerlis**. Safeslab: Mitigating Use-After-Free Vulnerabilities via Memory Protection Keys. October 2023.

[38] Marius Momeu, Fabian Kilger, Christopher Roemheld, Simon Schnückel, Sergej Proskurin, Michalis Polychronakis, and **Vasileios P. Kemerlis**. Immutable Memory Management Metadata for Commodity Operating System Kernels. August 2023.

[39] Meghna Pancholi, Andreas D. Kellas, **Vasileios P. Kemerlis**, and Simha Sethumadhavan. Timeloops: System Call Policy Learning for Containerized Microservices. September 2022.