Software correctness and safety relies on an accurate understanding of hardware behavior. Unfortunately, the abstracted interface that hardware provides often does not capture the full extent of its visible behaviors. Side-channels, speculative execution attacks, fault attacks, performance pitfalls, and more provide ample evidence that if software wants to be secure, it cannot ignore what happens “under-the-hood” when the abstraction leaks. My research interests are in building secure and trustworthy systems in the light of abstraction failures. Computer science relies on a highly successful strategy of layered abstraction from transistors to ISAs to software APIs. Because of this, my work encompasses all of these layers: spanning systems, web security, and hardware architecture.

My published work illuminates both new ways that the hardware abstraction can break down with practical attacks (caches [11], floating point timing [2, 3]) and builds new security-focused systems with the benefit of hindsight (browsers [4], Trusted Execution Environments [5], entropy generation [6]). My projects are practical and have had wide impact inside and outside of academic research. For example, my work has led to numerous changes to web browser defenses [7, 8, 9, 10, 11], have been used in verification projects [12, 13, 14], have been adopted by mainline kernels [15, 16, 17], and created new community projects [18].

Here, I summarize my defensive systems work, my novel attack techniques, and outline future directions.

Building trustworthy systems

Much of my recent work has built new, practical systems to resist adversaries with capabilities beyond a classical software attacker. The result is a set of work that crosses the stack, finding the right place to solve the given security challenge. These projects have had significant impact: causing major changes to how time is handled in web browsers, attracting verification efforts from academia, and driving adoption by industry. I build these systems to be verifiable, audit-able, and responsive to novel threats as they appear. In all of my projects, I built defenses with performance/security trade-offs to allow for concrete use-cases with different threat models. Here I detail my efforts in web browser security, novel trusted execution environments, and a timing-safe math library.

Trusted Execution Environments for diverse use-cases: My most recent project has been building the systems and community for Keystone [19] – an open-source framework for customized Trusted Execution Environments (TEEs) on unmodified RISC-V. Widespread availability of commercial TEEs like Intel’s SGX and ARM’s TrustZone has opened up new models for secure computation, trusted remote deployments, and minimization of trust. Already, TEEs can be found in every smartphone and many desktop CPUs. Soon, some form of TEE will be present on all commercial processors. This is a huge opportunity for trustworthy computation: when programs are run in these TEEs, even powerful adversaries with OS-level privileges or physical access to the machine can only affect availability, not integrity or confidentiality. Unfortunately, current commercial TEE designs have been found to be vulnerable in systematic ways, are closed-source, and have been slow to iterate due to being mostly built in hardware and microcode. Additionally, both commercial and academic TEEs tend to be single-purpose designs which makes it difficult to explore the design space of threat models and functionality trade-offs.

We solve these problems with the Keystone project: an open-source software framework that provides a new, far more flexible model for building TEEs. Keystone provides a software framework that can instantiate a specialized TEE for a given use-case on unmodified RISC-V hardware. Keystone is both a proving ground for TEE defenses/designs, and it serves as the basis for future industry products. Keystone is built to be portable to nearly any RISC-V system and provides the strongest guarantees possible for that given platform. This portability allowed us to experiment on in-order and out-of-order RISC-V cores without changes. Ongoing work allows Keystone to run on application cores (i.e. smartphone-class) and microcontrollers.

Keystone supports strong side-channel defenses I personally designed and developed. These defenses re-purpose commonly available hardware features to allow an enclaved application to execute entirely on-chip, without having code or data reside in DRAM. This is possible in Keystone without the need for additional custom hardware (as Intel SGX uses) because of our software-centric design. The defense is entirely transparent to the application and support for this type of feature is a compile-time option for our high-privilege
security monitor component. To establish trust that the defense was enabled, the remote user is provided with a cryptographically verifiable report detailing the defensive configuration. The flexibility I’ve designed into Keystone’s core allows us to freely experiment with novel hardware changes and software stacks on a common platform with common benchmarks. Building Keystone required an enormous development effort, including a custom hypervisor-like security reference monitor, a custom modular microkernel, numerous user-level libraries, SDKs, development tools, and significant documentation. We’ve accomplished this with a small core team of two post-docs and one PhD student.

To help expand the project, I interviewed, recruited, and co-advised a group of seven undergraduates at UC Berkeley. These students have made significant contributions to Keystone over the past year, notably in additional SDK tooling and a complete Rust rewrite for our secure reference monitor.

Keystone has already had significant success in attracting external interest. More than ten academic projects [20, 14] have cited or built on Keystone pre-publication. We’ve given over a dozen invited talks at universities, companies, and conferences. Keystone has attracted international interest and contributions [18]. Keystone has a rapidly forming community of academic researchers from a variety of universities and industry partners prototyping product integrations. To help foster that community and standardize approaches, I have co-organized two highly successful annual workshops [21] for industry and academia on open-source secure enclaves. These workshops included over a hundred attendees and dozens of speakers, including from Google, Seagate, HTC, Nvidia, Intel, ARM, and others.

Timing-attack resistant browsers [Oakland’16]: Timing attacks against user privacy in web browsers, such as to steal private user data and compromise accounts, have a long history of success with a wide variety of mechanisms. Motivated by my own experience writing a novel series of such attacks, we proposed and developed a timing-attack resistant browser: Fuzzyfox [4]. Without knowing what novel timing-attack the adversary has discovered, my design verifiably mitigates all attacks and completely prevents many of the most potent ones. This is accomplished by changing the way that JavaScript perceives time, restricting the adversary to only a coarse-grained view of when events occur. To do this, I reorganized the browser’s execution of events into time intervals during which no clocks in the system update. By then randomizing the start and end of these intervals slightly, we can control the granularity at which the attacker can measure time. I created Fuzzyfox by modifying the core event loop of Firefox to separate time into discrete epochs, during which the browser defers new updates and events until the next epoch. Within the event loop these epochs are enforced by synthetic event queues via system scheduled delay events. The net effect of this is that, no matter what, the rate at which an adversary can leak information is bounded to a known rate. Further, this leakage rate is naturally tunable for a performance/security trade-off. This defense can thus be strengthened at the cost of performance when a new vulnerability is found or for users who are high value targets.

I evaluated many previously developed attacks and timing techniques and found that they no longer worked against Fuzzyfox. In addition, I was able to experimentally verify that all previous techniques I had developed were now capped in their information leakage rate, as expected. Fuzzyfox is available on github [22] and has been used for comparison by several other security research groups.

In the wake of the Spectre-type vulnerabilities publicized in early 2018, Edge, Firefox, and Chrome all applied the same time fuzzing techniques to their explicit clocks. Had these mitigations been applied when I proposed them, the impact to web browsers from Spectre-style vulnerabilities would have been significantly decreased. Since then, Mozilla has re-written the Fuzzyfox prototype into Firefox [14] as an optional feature.

libftfp [Oakland’15]: As part of our work on floating-point timing attacks [2] we developed the only fixed-point constant-time math library, libfixedtimefixedpoint (libftfp), as a safer alternative to hardware floating-point. This library is written in C carefully designed to coerce compilers into generating constant-time code. It supports customizable partitioning of its 64-bit internal type at compile time and implements fixed-point versions of most SSE floating-point operations. Constructing this library served two purposes: to make available a non-integer constant-time math library and to demonstrate the impossibility of assured constant-time behaviors. No x86 processor vendor will guarantee that the limited instructions we use are safely constant time. The advent of ARM’s Data Independent Timing (DIT) extensions means a libftfp port to ARM could finally overcome this hurdle. libftfp is open-source [23] and was independently proven to be constant time when compiled to LLVM bytecode [12] and to be cache side-channel free [13].

Breaking hardware abstractions

Much of my thesis work was on finding assumptions software makes about hardware and demonstrating the resulting security failures. Without a through investigation into how the hardware abstraction can break, we cannot be confident in the security we build on top. My work on attacks thus serves as the motivation for my defensive systems work detailed above. The attacks below represent entirely new methods of attack,
not simple improvements to state-of-the-art. Here, I detail a long-running set of attacks using floating-point instruction timing variation and a novel cache attack leveraging a (at the time) new Intel feature.

**Floating-point timing attacks [Oakland’15, USENIX’17]:** Timing side-channels have shown their value in numerous scenarios, from information extraction on local devices to network-based cryptographic attacks. These are removed by writing constant-time code: eliminating branching code paths and normalizing memory accesses to avoid cache effects. In 2015, we published the first timing attack for commodity processors using only the execution time variance from changing arguments to an individual x86 instruction: an instruction-data timing attack. Our technique [2] was not confined to contrived circumstances, and we completely dismantled privacy protections for web browsers in-the-wild. This type of timing attack is fundamentally different from those previously known. Rather than relying on diverging code paths or changes in cache state, our attack is able to use minute differences in the number of cycles individual instructions take to complete. Vulnerable programs are thus nearly impossible for a programmer to identify; they appear at even the binary level to be safely constant time. The specific timing variation we document arises from floating-point unit (FPU) behavior during handling of different IEEE-754 floating-point value types in Intel and AMD processors.

Our main application of this technique was attacking browser privacy and security, which is heavily reliant on the Same Origin Policy (SOP). SOP at a high level states that one web origin should not be able to unilaterally read information about another web origin. This is part of what stops a malicious web page from embedding resources from, say, a banking website and reading out account information. To break SOP, we used an SVG-filter timing attack rebuilt for our floating-point timing channel. This style of attack used a combination of JavaScript and SVG filters to read individual pixels across origins, violating SOP. Our novel variant constructed an attacker-controlled web page where a series of floating-point divisions, multiplications, and additions occur between a secret value (a pixel color value from a victim web origin) and an attacker controlled value. This exhibited enough timing variation for us to build a timing side-channel attack that steals usernames, login information, and web browsing history. While there had been a prior series of fixes to Mozilla Firefox to specifically address timing side channels in SVG filter implementations, they had not fully anticipated the threat posed by the individual arithmetic operations themselves. We also showed that academic security researchers were unaware of the danger these timing variations posed, and we demonstrated a timing side channel attack against a database specifically designed to keep data safe from timing attacks.

I followed this initial Firefox attack with a more detailed analysis [3] of the underlying mechanism and presented new attacks for deployed defenses by major browser vendors. By making subtle changes to how the attack functioned, I was able to demonstrate complete breaks of privacy for many major browsers, resulting in high profile bugs and a bounty payout from Chrome. (See Mozilla Firefox CVE-2017-5407, Google Chrome CVE-2017-5107, and Apple Safari CVE-2017-7006.) Based on my work, all browsers have since deployed new defenses to close the side-channel. I am most excited about the changes only Safari was willing to deploy; based on my proposal, Safari reworked SVG rendering to remove mixing origins in SVG rendering, permanently closing the attack vector.

**Evasive and more accurate cache attacks [USENIX’17]:** Understanding the exact details of how processor features or performance trade-offs are made is critical to software security. As new features are introduced to processors, they always have the potential to enable new avenues of attack or defense. We developed an entirely new class of cache side-channel attack that bypassed many existing defenses using such a new feature: Intel’s Transactional Synchronization eXtensions (TSX). We found that TSX’s failure cases, with careful setup, can expose subtle information about the state of shared caches. Our resulting technique, Prime+Abort [1], is a uniquely timer-free cache side-channel. To make our attack work, we first load a specific set of cache lines into the read or write set of a TSX memory transaction. At this point, Prime+Abort will receive a callback whenever victim activity disturbs those specific cache lines. Unlike other cache attacks, we do not rely on any timing information about accesses and do not require any measurement period to determine the state of the cache. As if to illustrate the unexpected outcomes of hardware features, at the same conference we presented Prime+Abort an independent group presented a cache timing-attack defense using the same TSX mechanisms. In 2019, likely in response to several years of attacks utilizing TSX, Intel released microcode patches disabling portions of TSX entirely. It is our belief that had this feature been more carefully developed, it would not have been such a boon to attackers.
Future research

My future, and current, research matches the path of my previous work: identify new ways hardware abstractions fail and then use my understanding of the underlying causes to build safer software-hardware systems.

**Hardware reliability and security assumptions.** My work has identified places where specific performance characteristics or details of feature implementation can leak to undermine high-level software security guarantees. It is not yet clear how to avoid the next new feature enabling new attacks, like happened with TSX. Solving this has two parts: collaboration with hardware architects to evaluate new hardware changes as they happen, and proposing new ways for hardware to more safely deploy features. As part of Keystone, I actively participate in the RISC-V standards working groups to help guide RISC-V in directions more valuable for security.

**Challenges in using FPGAs securely.** FPGAs are rapidly becoming available in both commodity processors and cloud environments. A problem the Keystone project has faced is the lack of secure FPGA programming. There are no ways to establish trust that a remotely programmed FPGA was imaged correctly. This results in FPGAs being unusable for deploying secure hardware designs and are only useful for prototyping. FPGAs are also a highly valuable avenue for side-channel attacks, and significant work is required to have them be safe for cloud deployments.

**Rethinking the ISA abstraction** The recent trend of Spectre-class speculative vulnerabilities has made it clear that the current way we specify the hardware-software contract is broken. How software can better communicate its expectations about hardware behavior is an open problem. I believe there is promise in moving the agreement on a contract to runtime, rather than compile-time, to allow for the same level of hardware innovation without sacrificing software security. This is similar in concept to the way that enclaves may request specific features at-runtime in Keystone, but extended to use-cases beyond TEEs.

**Designing attack resistant hardware and software.** Keystone, Fuzzyfox, and libftfp are practical examples of the types of defensive systems I’ll continue to build. A clear next objective is building hardware and software systems that have provable timing properties in composition. The ability to prove timing characteristics of hardware will allow our existing software tools make fully justified assumptions about the underlying hardware implementation. Keystone provides a testbed for my future work on hardware changes and new software stacks in this space.
References


