I work at the intersection of systems, networking, and security. My research focuses on practical solutions to fundamental open problems in these areas, coupling understanding of all layers of the software stack and hardware. I have designed and implemented novel systems that enable energy-efficient, privacy-preserving, and secure mobile computing. While clean abstractions and designs are necessary, they are not sufficient; I believe it is equally important to demonstrate that they can be reduced to practice. Towards this end, many of my projects entail large development efforts, including significant modifications to open-source operating systems. Not only do these implementations aid in proving the thesis of a given research work, but they are fundamental in refining the higher-level ideas and designs.

Next, I discuss my work on three different projects that illustrate the span of my research. The first two projects address problems in power management and security for mobile systems, while the third project focuses on primitives for privacy-preserving mobile social communication. In addition to these projects, I have also worked on others that involve topics such as network measurement [4], wireless spectrum allocation [7], censorship-resistant communication, and power management for multi-battery systems. Finally, I conclude with a discussion of the directions I have planned for my future work.

Operating System Power Management

Drowsy [5]: Drowsy is a novel approach to improving the energy efficiency of smart devices that perform (periodic) short-lived tasks in the background. As a result, Drowsy also enables applications that were not practical before without specialized hardware, such as continuous and context-aware sensing.

Portable smart devices are energy limited, requiring the design of hardware and software to consider energy efficiency as a first order goal. A common technique is to employ an OS-provided, system-wide power management sleep state, whereby the CPU and all devices are placed in low-power modes with their state stored in memory (which retains its contents). For example, Android defaults to the sleep state and applications must explicitly retain locks to remain in the on state. These deployed power management techniques were designed for human-driven wakeups, which are infrequent and tend to last on the order of minutes (or longer); however, there are many (periodic) short-lived wakeups that execute in the background on mobile devices, such as: receiving push notifications, sampling sensors, and communicating with nearby devices. In fact, we first discovered this problem after analyzing the energy efficiency of our SDDR [3] protocol (discussed later) which periodically communicates with nearby devices. For such wakeups, the energy spent transitioning between the sleep and on states is inordinate relative to the energy required to handle the event itself; for example, transitions account for roughly 75% of the energy consumption in waking up to fetch a weather update from a remote server. In analyzing these transitions further, we determined that the vast majority of energy consumed results from suspending or resuming all tasks and devices.

We designed a new power management state called drowsy, which acts as a direct replacement for the on state. In transitioning from the sleep to drowsy state, the operating system resumes only the necessary tasks and devices required in handling the wakeup event. While this is a conceptually simple abstraction – “just wake up what is needed” – part of my work is to demonstrate that a practical implementation is possible. In practice, there are many complex inter-dependencies between software and hardware that must be accounted for in order to maintain functionality. A key constraint we imposed on our design and implementation was to not require any modifications to user-space applications, thereby enabling the deployment of Drowsy on any device by simply patching the OS kernel. Drowsy constructs the minimal set of tasks and devices to resume over time as the event progresses by inferring and tracking dependencies between devices and tasks at runtime.

We augmented the Android/Linux kernel with support for Drowsy by instrumenting all of the interaction interfaces that expose such dependencies between devices and tasks (e.g., sysfs file operations, interrupts). Applying our implementation to the Nexus 4 smartphone, we evaluated the real-world improvements in energy efficiency across a wide variety of micro- and macro-benchmarks. Our results demonstrate that Drowsy improves energy consumption by 1.5-5x for common short-lived tasks; as shown in Figure 1a, Drowsy reduced the energy consumption by a factor of 3x for fetching data from a remote server (a weather update). Ultimately, this is a remarkable result in that the kernel is fully functional and improves the energy consumption for a highly optimized OS that is deployed on more than a billion devices. This work was published in SOSP [5].

Matthew Lentz (mlentz@cs.umd.edu)
Securing Sensitive User Data on Smart Devices

Personal smart devices provide users with powerful capabilities for communication, productivity, health, education, and entertainment. Applications often operate over sensitive I/O data related to the user: collecting and processing input data from sensors (e.g., fingerprint scans, location updates), or rendering output data to the user (e.g., display financial information). We address the problem at the level of I/O because, no matter what, I/O devices always serve as sources and sinks for the sensitive data. This sensitive data is the target of many attacks, which range from malicious applications to compromises of the platform software itself. Today, users are ultimately unable to control or reason about how their sensitive data is processed, protected, or shared.

SeCloak [6]: SeCloak augments existing smart device platforms with support for secure, virtual switches that allow users to reliably express on/off control over I/O devices. SeCloak acts as a separate, platform-agnostic layer that enforces the user’s control policy, and thus does not require trusting the existing platform software (which has proven hard to secure).

In our initial work, we focused on addressing a single (yet important) point in the space of providing users with control: giving users on/off control for I/O devices. For example, journalists interviewing confidential sources may want to use the microphone but reliably turn off the radios and GPS, and users may want to make sure the camera and microphone are disabled during private meetings. The settings offered by existing platforms are often insufficient (e.g., lack of accelerometer protection on Android) or inadequate (e.g., iOS Control Center disconnects, but doesn’t disable, WiFi and Bluetooth). Additionally, users have little to no assurance that the settings are enforced because existing platforms are insecure as a result of their complexity and large attack surface.

Our goal was to provide a simple abstraction to users of secure virtual switches that they can use to control the various peripherals on their device. We designed SeCloak to implement this abstraction, which acts as an independent, platform-agnostic enforcement layer beneath the existing platform and applications. In order to minimize the trusted code base, we divided the design of SeCloak into two parts: the user interacts with an untrusted settings application that communicates the desired policy to the enforcement layer, which is responsible for requesting explicit user confirmation before applying and enforcing the settings based on a trusted specification of the hardware. In our implementation, we leverage ARM TrustZone hardware security extensions, which provide hardware isolation between two “worlds” of execution (non-secure and secure). We run the existing software base (e.g., Android/Linux) and our untrusted settings application in the non-secure world, while our enforcement kernel operates in the secure world. To handle cases where the protections are too coarse-grained, as well as to securely share several devices with the non-secure world (e.g., hardware buttons), the secure kernel traps the faults generated when the non-secure world accesses protected devices and selectively emulates the accesses. We implemented the entire secure kernel of SeCloak for the i.MX6 SoC, and our evaluation demonstrated the minimal performance overhead incurred when emulation is needed. This work was published in MobiSys [6].

Matthew Lentz (mlentz@cs.umd.edu)
Accountable Paths [2]: Accountable paths allow users to express control over the collection and processing of data on their devices, and enable reasoning about (and enforcing) assurances over the data along end-to-end paths between applications and the underlying I/O devices. The guarantees provided by accountable paths are decoupled from the correctness of the rest of the platform software.

SeCloak addresses an important problem of enabling users to dictate on/off control policy. It is, however, a relatively blunt tool that doesn’t support more broad forms of expressing control; for instance, allowing users to control the resolution of GPS sensor samples instead of simply denying access. We still need to address the inability of users to reason about the software handling their sensitive data, from the application down to the hardware devices, and what (if any) assurances it provides (e.g., confidentiality). Both of these problems extend beyond the users themselves to other stakeholders in the ecosystem; for example, a bank may require assurance that the user explicitly confirmed a monetary transaction.

We introduce the accountable path primitive, which enables fundamental reasoning about the software that processes I/O data, while providing first-class support for expressive control over the I/O data and processing. An accountable path represents a dynamic binding between software modules for invoking some I/O functionality, which is subject to certain assurances (i.e., confidentiality, integrity, and availability) and trust assumptions (e.g., “rely on modules from X developer”). Each accountable path may be naturally be composed of multiple lower-level accountable paths; for example, providing an endpoint for requesting PIN input from the user will rely on paths for interacting with the touchscreen and display devices. The module that constructs an accountable path may later invoke it and also request attestations over the state of the path. We support fine-grained delegation of trust and privileges for principals (and their modules) based on a credential scheme backed by a trusted specification of the hardware. More-privileged principals, such as the user themselves, may affect the interactions and data on paths constructed by less-privileged principals by applying programmatic policies.

We designed and built a version of the AIO platform, which implements the accountable path primitive, on top of the ARM TrustZone security extensions. AIO mediates all hardware access and all inter-module communication to enforce necessary access control and other applied policies. Using AIO, we augment the existing Android/Linux platform with support for accountable paths and demonstrate how applications can take advantage of this support. To aid in application development, we provide a toolbox containing modules that serve as building blocks for developers to use (or even extend/replace); these blocks range from a PIN input interface for user authentication to a proxy for modules running on remote servers. This work is currently under submission [2].

Encounter-based Communication

Encounter-based communication enables a wide variety of mobile social applications based on addressing “encounters”, which represent periods of co-location between users (and their devices).

Many mobile social applications provide services to users based on their context, such as: detecting presence of friends, sharing recommendations for places close by, and exchanging content with nearby users or those previously encountered (e.g., missed connections, lost-and-found). At their core, these applications rely on support for discovering nearby devices, communicating between currently (or previously) co-located devices, and (optionally) mapping discovered devices to known users. Many deployed solutions rely on trusted cloud services, presenting a privacy risk to users. Instead, we focus on providing a secure encounter abstraction, which is based around a period of co-location between pairs of users (and their devices). Our work in this area includes designing a privacy-preserving and energy-efficient secure encounter protocol (SDDR [3]), as well as crafting higher-level abstractions on top of secure encounters that applications can use for group communication (EnCore [1] and enClosure [8]).

SDDR [3]: A secure encounter protocol generates a shared secret between two co-located users, which can be used to establish a secure communication channel. Users participating in a secure encounter should be “unlinkable” by default, such that no identifying information is exchanged; however, at the same time, selective linkability of known users should be supported along with efficient, unilateral revocation. We can construct a protocol using existing, standard cryptographic tools: Diffie-Hellman (DH) key exchange to generate a shared secret, and Private Set Intersection (PSI) over sets of secrets exchanged between linkable users. While this protocol is both correct and secure, it is not practical to run continuously on resource constrained mobile devices (as shown on the top panel of Figure 1b).
In designing the Secure Device Discovery and Recognition (SDDR) protocol, we treated energy-efficiency as a first-order goal along with satisfying the privacy and security requirements. Key to this was constructing SDDR as a non-interactive protocol, allowing us to leverage a Bluetooth feature whereby the controller can directly respond to discovery requests without waking up the application processor. We formally prove the correctness of our non-interactive SDDR protocol. Our results show that SDDR is practical, consuming only 10% of the battery capacity of a typical smartphone when run continuously throughout a day; as shown in Figure 1b, SDDR is extremely energy efficient (by a factor of 19x over the standard protocol). This work was published in USENIX Security [3].

**Encore** [1] and **enClosure** [8]: SDDR is an essential low-level protocol that establishes secure encounters between users; however, we need to provide a higher-level abstractions for useful communication. We focused on this problem in our work on building Encore and enClosure, both of which were published in MobiSys [1,8].

EnCore introduces a new communication abstraction based around *events*, which represent groups of proximal users and are associated with inferred context and user annotations. To exchange messages between event participants, EnCore leverages existing services that can support communication, such as remote storage (Dropbox) and online social networks (Facebook). We performed an extended user study by conducting a series of live deployments with 35 users at the Max Planck Institute for Software Systems, providing users with an application built on EnCore to enable them to construct events and use them to communicate, share, and collaborate. Results from our study demonstrated that users were engaged with the application and found the paradigm useful, while using it in a number of new (and unanticipated) ways.

In enClosure, we leverage the *encounter graph* which is formed by globally aggregating all individual pair-wise encounters; nodes correspond to users and each edge represents an encounter between a pair of users. To enable new classes of communication for peers beyond those directly encountered, we introduce a new abstraction for addressing communication that consists of specifying spatial, temporal, and/or causal constraints over the encounter graph. Each device periodically uploads its local encounter database to the cloud-based forwarding agent, which uses its global view to determine the closure over the set of encounters to forward the message to based on the address constraints. To protect the confidentiality of encounter data, we implement and run the forwarding agent within a trusted execution environment (e.g., Intel SGX enclave), while communication takes place through an untrusted key-value store. We demonstrate the utility of enClosure by implementing several applications (e.g., virtual guest book and health-risk warning) which we evaluate using a synthesized encounter graph derived from 6.4 million check-ins for users of the Gowalla social network over the course of nine months.

**Future Directions**

In future work, I plan to expand my research in three major new directions, namely: 1) employing formal methods to strengthen the foundations of my work on secure systems, 2) leveraging machine learning for implementing unobservable, censorship-resistant communication, and 3) enabling scalable and interactive live streaming platforms. I discuss each of these in turn.

**Building Higher Assurance Systems** A key goal of my work on accountable paths is to provide assurances over the handling of sensitive I/O data; however, these assurances naturally rest on any assurances over the correctness and security properties of the AIO platform that implements accountable paths. While designing a narrow interface, minimizing the trusted computing base (TCB), and due diligence in writing and testing the implementation are useful, we can do better. Formally verifying parts of (or the entire) design and implementation provide a much higher level of assurance. In recent years, there have been many advances in practical formal methods that can be used to prove correctness and security properties for larger systems. However, there is still a significant gap, and I would like to look into bridging it as part of my research agenda. For example, my work on accountable paths can directly benefit from such formalism, and I plan to investigate an implementation on top of the formally-verified seL4 microkernel. In particular, I believe it is possible to construct an efficient implementation that casts accountable path operations onto the strict capability system provided by seL4. In doing so, we would be able to leverage the existing, formally-verified model of these capabilities to serve as the basis for our own proofs of correctness and security guarantees for the accountable path platform.
Machine Learning for Censorship-Resistant Communication In ongoing work, I am aiming to enable high-bandwidth, unobservable, censorship-resistant communication by utilizing uncensored video chat applications as the communication medium. In order to tunnel Internet traffic through the (unmodified) video chat application, our system poses as a webcam that encodes the packet data into the raw video frames it outputs. Much like physical layer wireless links, the transmitted data is subject to errors as a result of channel effects; in our case, these channel effects correspond to the lossy video encoding/decoding operations over the frames (along with underlying network losses). To remain unobservable, we must output video frames such that the traffic over the network cannot be distinguished from that of “normal” traffic (e.g., someone sitting in front of a camera). We are investigating the possibility of using generative adversarial networks (GANs) to build a model for mapping packet data to video frames by training on typical video chat content; on the other end, the receiver uses the inverse of the generator to recover the original data.

More generally, I am interested in understanding the constraints and issues facing deploying of machine learning models in real systems. There is much work, fueled by the broad success of machine learning, that focuses on applications to many different scenarios (e.g., replacing heuristics in existing systems). While such work is obviously useful, I am most interested in answering some more fundamental questions. For instance, despite the success of many modern machine learning techniques, most of the complex models are not explainable. How can we apply these models to cases like communication systems, whether over video or more-traditional wireless mediums, and still maintain assurances? What abstractions would be useful to address these issues?

Enabling Scalable/Interactive Live Streaming Platforms In recent years, live streaming services like Amazon’s Twitch and Microsoft’s Mixer have grown in popularity, enabling users (or streamers) to share live content with their viewing audience, ranging from hosting talk shows to playing video games. A key aspect of these platforms is that they enable the viewers to interact with the streamer, along with one another, through real-time chat rooms. This is an interesting problem space since there are a number of practical challenges that exist in building such platforms in a scalable way. For example, one such challenge is in scaling support for live transcoding to all streamers, which helps reach viewers with bandwidth and/or device limitations that would otherwise. Currently, these services provide transcoding for a limited number of “partnered” streamers, while the number of new streamers continues to steadily rise. Can we do better to support the democratization of streaming? Is it possible to shift most of the transcoding effort off of the platform itself and onto the streamer, without significantly increasing the bandwidth and computation requirements for the streamer? Can we consider a minimal restructuring of the encodings for popular codecs such that lower-resolution encodings are always prefixes of higher-resolution encodings (analogous to the rateless coding property)?

References