RESEARCH STATEMENT
Kristopher Micinski (micinski@cs.umd.edu)

Determining whether or not a system is secure is inherently contextual, informed in part by human judgment. For example, a user will likely be okay with their phone using their camera immediately after they click a button to take a picture, but will likely be upset if the camera is subsequently used later without telling them. As our modern software systems become increasingly complex, it becomes challenging to connect the behavior of these systems to the informal notions of security in users’ minds. To be confident our systems conform to some consistent definition of security, we have to appeal to formal techniques when we reason about those systems. But to be confident that the systems we build are meaningful to users, we have to evaluate them by studying humans.

My research reconciles these two challenges by combining programming languages—which allows us to formally define what a system is doing—with human-computer interaction, which allows us to understand how users conceptualize security. I start by building tools to help experts reason about what production systems are doing. This allows me to understand what security should look like for those systems. Using those insights, I assess the gap between formally specified notions of security and the preexisting frameworks that secure those systems. I then work to mitigate that gap by integrating formal approaches into those systems. These formal approaches give ground truth to some objectively defined notion of security. But humans offer ground truth for usability. Concurrently, I run experiments with users to understand their perceptions about what security should mean for these systems. This establishes a feedback loop between both usability and formal approaches. This allows me to build systems that allow users to understand security on their own terms, while being backed by formally established guarantees.

Throughout my dissertation work, I studied security in mobile apps. Because mobile apps have access to a large amount of sensitive user data, it is crucial to ensure that users are aware when that data is accessed or declassified. My insight was that the GUI served as a bridge between how users interact with apps, and how we can formally define and enforce security policies. I used this insight to define a new class of policies that leverages the app’s GUI to enforce informed consent. Going forward, I plan to expand these techniques to other domains and across platforms. I will continue to leverage techniques such as program auditing and scalable program analyses, which allowed me to apply these formal ideas in security to production systems. Applying those ideas, I will build systems that harmonize user experience with rigorous notions of security. Last, because my work frequently involves a mix of building systems and formal reasoning, I envision rich collaboration and mentorship opportunities with students at all levels.

Dissertation Work

Binary Rewriting to Retrofit Policies in Production Apps I began my dissertation research by extending the Android permission system to offer enhanced security on unmodified devices. Android protects sensitive resources (such as location) with permissions: per-app tokens that are required before sensitive data may be accessed. However, if users don’t want to accept the permissions the app requires, they are left no recourse but to not install the app. As a remedy, my collaborators and I implemented Dr. Android and Mr. Hide[1]: a trusted service (Mr. Hide) to arbitrate access to sensitive resources and a binary rewriter (Dr. Android) that retrofits apps with a desired security policy [4].

For an example use of Dr. Android and Mr. Hide, consider an app that accesses the Internet. The user may be comfortable with the app accessing a domain such as cnn.com, but uncomfortable accessing domains such as evilads.com. To implement this policy, the user would specify that they wanted to restrict the app to only access cnn.com. Then, Mr. Hide would access the Internet on behalf of the app. This is done by Dr. Android, which redirects networking-related API calls through Mr. Hide (via inter-process communication) instead of the regular Android APIs. When Mr. Hide receives requests to the Internet from an app, it makes sure the app is allowed to access the specified domain, and if not Mr. Hide silently drops those requests.

We applied Dr. Android and Mr. Hide to 14 apps from Google Play and was successfully able to enforce fine-grained permissions access for each of those apps while imposing only a modest overhead. This was all performed on stock devices without modifying the Android operating system. With the help of undergraduate student Rebecca Norton, I publicly released Dr. Android as a more general binary rewriter for Android named Redexer[2]. Redexer is

---

1Dr. Android is the Dalvik Rewriter for Android (Dalvik being the bytecode language of Android) and Mr. Hide is the Hide interface to the dalvik environment
2https://github.com/plum-umd/redexer

http://cs.umd.edu/~micinski/application
used by a variety of researchers to enhance security on Android. To date the Dr. Android paper has been cited 140 times (180 including citations to the tech report version).

Throughout my work implementing finer grained permissions, I realized that many apps used location information, even when not obviously performing tasks on behalf of the user (such as location-enabled search). I worked with a high school student (Philip Phelps) to build an extension to Dr. Android and Mr. Hide that allowed the user to truncate parts of the user’s location before giving it to apps. Doing so offers a cheap form of partial location anonymity. However, we wondered whether our system would affect the user experience of most apps using location-based services. To measure the effect of location truncation on apps, we varied the amount of truncation performed and evaluated the app’s output in a variety of locations. To do this, we built a systematic testing tool that sent apps mock input sequences and then captured the state of the GUI. We then measured the impact of truncation on usability by measuring metrics such as edit distance on location-enabled search results [6]. We found that the ability to truncate location depended on the population density of the surrounding area, but that many apps tolerate between 5 and 20km of location truncation and still present roughly the same results to the user.

Hyperproperties and Temporal Logics for Information Flow Many interesting statements about a program’s security are not properties of a single run of that program. For example, noninterference is a formal way of stating that a program does not leak any secret input. A program satisfies noninterference if any run of that program will be equivalent (from the perspective of a public observer) to a run with the secret inputs replaced by any other value. This makes noninterference a property of pairs of runs. Many of these so-called information flow statements quantify over multiple runs of a program simultaneously, and fall into a class of program properties called hyperproperties [2].

Because many useful programs leak some information about their secret inputs, a long standing problem in information flow is how to reconcile declassification in a principled way. To solve this, my collaborators and I designed two logics that integrate temporal logics and hyperproperties: HyperLTL and HyperCTL [1]. To evaluate the usability of these logics to secure systems, I built a model checker for a fragment of HyperLTL and ran it on a set of labeled transition systems. I was successfully able to automatically verify these systems satisfied various formulas in HyperLTL that enforced security statements related to noninterference.

Symbolic Execution and Interaction-Based Declassification Next, I attempted to apply information flow policies to Android apps. This was challenging because almost all Android apps requesting access to some resource violate noninterference. Implementing declassification is challenging in real systems because it is frequently unclear how to inform the user when declassification happens. The key insight I had was that apps use the GUI to implement informed consent: secret inputs can be declassified only when the user explicitly performs some GUI interaction. Based on this idea, I defined interaction-based information flow. I formalized this by defining a variant of noninterference and applying it to Android apps using program analysis [5]. In my work, special interaction-based policies connect GUI events to the information they declassify. An example interaction-based policy is shown below:

\[ \text{ph!} \ast \land (F(\text{sendBtn!unit} \land \text{last(phBox, true)})) \Rightarrow \text{Low} \]

Here, the ph! notation is used to represent a write of any value on the channel ph, which is used here to represent the phone number. The policy can be read as follows: “At time \(i\), if the input at time \(i\) is on the ph channel, and at some future \(j\) (indicated by the stylized \(F\)) the send button is pressed \(\text{sendBtn!unit}\) while the last setting of the phBox was true, the input at time \(i\) may be declassified to security level Low.”

Because these policies rely on precise temporal orderings of program states, I used symbolic execution, a program analysis that executes the program with symbolic variables. In a previous collaboration I worked on SymDroid [3], a symbolic executor for Dalvik bytecode. I extended SymDroid to create ClickRelease. ClickRelease checks these interaction-based policies by collecting pairs of program paths and forming equations over them. These equations encode interaction-based declassification to guarantee that apps only leak data in accordance with the policy. I applied ClickRelease on four synthetic apps including benign and (two) malicious variants of those apps. ClickRelease was able to find information flow leaks in the malicious variants and generate counterexamples.

Comparing User Interactions and Security on Android After defining and enforcing interaction-based declassification, I wanted to understand to what extent such policies match user perceptions of security. In my most recent work, I study how user interaction informs user expectations about sensitive resource usage. I do this by conducting two studies: one that measures 150 top apps to determine the interaction patterns apps used to access permissions, and one that measures users to test how closely those patterns align with user expectation.

I first needed to understand the relation between an app’s GUI and how it accessed sensitive resources. Techniques such as symbolic execution suffer in that they do not scale to production apps with large codebases that make

http://cs.umd.edu/~micinski/application
extensive use of complex libraries. Instead, my technique works by using instrumentation to logs paths through the app as it is executed [7]. I implemented this in a tool called AppTracer. AppTracer inserts instrumentation into an app that logs its control flow. I can then run it on a device either manually or via automated exploration and capture an execution log. Next, AppTracer transforms this log of program behavior into an execution graph that helps an auditor understand which GUI events caused which permission uses.

I ran AppTracer on 150 popular Android apps to classify the types of interactive resource use in those apps. I used the graph AppTracer produced and a human-curated codebook to assign a set of codes to each permission use within each app. For example, in the graph on the right (a subgraph of one produced by AppTracer), the app read the contacts immediately after a button click, which is coded as a Click use of that permission.

Next, I conducted a user study. This study evaluated when users expected permission use to occur with respect to the app’s GUI. It is comprised of a walkthrough of an app interspersed with questions about what resources the user expects will be used. A partial example is shown on the left, wherein the user first sees an app’s home screen, then presses the “voice order coffee” button. To mirror the current Android permissions system, they are then prompted to allow access to the microphone — in other scenarios this dialog occurs at the beginning of the app or not at all to measure the effect of timing on user expectation. After seeing the dialog, the user is asked a series of Likert-style questions assessing their expectation the microphone will be used at that point in the app. The user is then shown another interaction with the app, in this case going back to the phone’s home screen. They are again asked expectation questions to understand how their perceptions of permission use changes as context changes.

Combining the results of our two studies we discovered that apps mostly matched user expectations, but noted some shortcomings with the permission system. Users seemed to expect the most invasive permissions (e.g., camera and microphone) to be used only after a click. Our app study confirmed this is almost exclusively the case. We recommend that this be made mandatory with rare exceptions, since uses of these resources not clearly associated with a relevant click are unexpected by users. Even when users understand resources will be accessed after these clicks, Android still asks users for explicit permission via a dialog screen. This implies Android is being too invasive, and that these interactions alone are sufficient to authorize the permission use. Also, we found that when users were prompted for permission immediately after a click, they assumed that that resource was used only after that interaction. For example, if an app waits until a map screen to ask users for location, users will be much less likely to expect the location to be used later in the app (perhaps, e.g., while the app is off the screen). We observed infrequent but occasional uses of this in our app study.

Future Directions

The work I did in my dissertation connects brings formal notions of security to users in mobile apps. But going forward, I plan to study these problems in a broader setting. For example, I believe my work can readily be applied to web apps or even across application platforms. However doing this will likely require foundational extensions to the techniques I relied upon during my dissertation work. These include program auditing, best-effort analyses, and visualization tools. I will leverage these techniques—along with my background in HCI—to attack new ideas that continue to bring high-assurance security to systems in a way users understand. I will briefly describe a few concrete directions, pointing out where they fit in to the overarching narrative of my work.

**Implementing and Explaining Cross-App Policies** Each application platform has its own notion of security policies. Unfortunately, many of these platforms are interconnected. For example, consider a mobile app that reads data from a social network and subsequently propagates that data to an ad provider. As users install more and more apps across each platform, it becomes hard for them to reason about their security. Although there has been recent work on analyzing systems to understand cross-platform flows, there is little work that helps explain these flows to users.

To address this problem, I plan to continue my recent work in program auditing and visualization. For example, I would likely start by using the infrastructure I built to log Android app behavior and expand that to include APIs from popular social networks. I would then survey apps to understand how they use and propagate information,
and interview end users to see what information they believe should be propagated across apps. Based on these discussions, I will design policies and tools for expressing them to users—likely drawing upon visualization. More long term, I see this work expanding to the setting of, e.g., web apps written in languages such as Ruby and JavaScript.

**Best-Effort Program Analysis via Partial Concretization** Reasoning about security frequently involves analyzing production systems for complex properties. But program analysis struggles to balance precision and scalability with correctness of results. Based on my work in program instrumentation, I envision using logs as a rich source of data to help assuage traditional problems in program analysis. Traditionally, static program analyses incur a high amount of uncertainty at some points in the program because they cannot fully decide runtime behavior. By contrast, dynamic approaches often fail to get adequate coverage of the program. However, I have observed it is often possible to leverage the dynamic behavior of the program to help inform a static analysis and get past the challenging parts.

For the short term, I plan to use instrumentation to help drive analyses such as symbolic execution. Symbolic execution traditionally encounters state explosion problems: as the exploration dives deeper into the program, exponentially more states are generated. To help get past this, I will use dynamic information from the program to decide which paths to explore. For example, one place that could cause a symbolic executor to get stuck is invoking a method on a symbolic object which could theoretically be any number of concrete classes at runtime but in practice is just one. I will use the dynamic logs of the program to help reduce the search space. Doing this trades precision for coverage. Understanding how to reconcile analysis completeness with these partially concrete techniques is a challenge I plan to address. Longer term, I believe this approach can be applied to other types of program analyses such as type systems and abstract interpretation.

**Visualization-Based Program Understanding** During my dissertation, I realized that one underutilized technique to understand programs was visualization. For example, program visualization technology has helped me more rapidly evaluate security policies for Android apps. I believe it will more broadly be useful for other security auditors, programmers, and students.

I have performed two pieces of initial work in this area. The first is AppTracer, which visualizes dynamic logs to help security auditors understand whether or not an app is secure. The second is a visualizer for the SymDroid symbolic executor, which augments the symbolic execution with a debugger-like interface. Going forward, I believe that visualization-based techniques will be crucial in understanding the results of my tools. Critically, I will use techniques from HCI to evaluate the strategies I develop and use that to guide the design of my tools.

**References**


