RESEARCH STATEMENT

Kristopher Micinski (kris@cs.haverford.edu)

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. I build on foundations from programming languages and apply them to security.

In my dissertation I studied security in mobile apps. Mobile apps have access to a large amount of sensitive user data. So it is crucial to ensure that users are aware when that data is accessed or declassified. My insight is that the app's GUI serves as a bridge between a user's conceptualization of security and an app's implementation. Using this, I define a new class of (so called *interaction-based*) policies that leverages the app's GUI to enforce informed consent. I built tools to measure these policies in apps and studied users to understand how they conceptualize security with respect to the UI.

My goal is to allow users to have precise control over their data with minimum overhead (cognitive load, technical understanding, cost, etc.). This requires further research in many directions. For example, it requires tools to help us understand what the policies should be. Then, it requires us to understand how to present these policies to users. If we want to be sure our programs behave according to these specifications, we also need tools to enforce these policies.

I still have a lot of work left to do. Much of my research includes scaling core ideas to implementations that work on production systems. I see this as an ideal platform for undergraduate research because it allows students to learn science in the context of novel scientific contributions.

Dynamic Analysis for Security To decide which problems in security to tackle, we must first measure app behavior in situ. I do this is using dynamic analysis. Dynamic analysis retrofits off-the-shelf apps with runtime tracking to check or ensure some property. My first collaboration—Dr. Android and Mr. Hide[1]—allows replacing permissions (e.g., INTERNET) with finer-grained permissions (e.g., cnn.com). This is done using 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 [1].

We applied Dr. Android and Mr. Hide to 14 apps from Google Play and were successfully able to enforce fine-grained permissions access for each of those apps while imposing only a modest performance overhead. This was all performed without modifying the Android operating system. With the help of my collaborators, I publicly released Dr. Android as a more general binary rewriter for Android named Redexer[2]. Redexer is actively maintained and used by groups in industry and academia. To date the Dr. Android paper has been cited 190 times (240 including the tech report version). After this work, I also worked with a high school student (Philip Phelps) to build an extension to Dr. Android and Mr. Hide that allowed the user to truncate bits of the location before giving it to apps. We measured this tool on a set of apps and published our results at Mobile Security Technologies (MoST) 2013 [4].

Information Flow and declassification Dynamic analysis is useful for tracking runtime properties of programs, but doesn't offer very strong formal guarantees. For example, dynamic analysis can't tell us anything about the paths we don't see at runtime. *Information flow* is a powerful property that can offer formal privacy guarantees. Informally, a program satisfies information flow if any pair of secret inputs leads to the same public output. But this power comes at a cost: it has proven hard to apply to realistic applications. I wanted to apply information flow policies to Android apps. The problem

[1]Dr. Android is the Dalvik Rewriter for Android (Dalvik being the bytecode language of Android) is the Hide interface to the dalvik environment

[2]`https://github.com/plum-umd/redexer`

was that most real apps leak *some* information. Various researchers have studied information flow in the presence of declassification. Previous efforts forced programmers to provide annotations. But this is unrealistic for typical apps. Further, it doesn't help users understand the policy.

My hypothesis was that well-behaving 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 information flow and applied it to Android apps using program analysis [2]. This work appeared at ESORICS 2015, a strong security conference with a 20% acceptance rate.

Comparing User Interactions and Security on Android I wanted to test my hypothesis. So next, I ran two studies to measure how the UI informs expectation of sensitive resource usage[3].

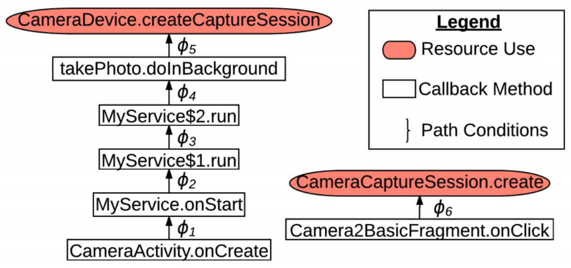
The first was an app study. I needed to understand the relation between an app's GUI and how it accessed sensitive resources. Techniques such as symbolic execution are hard to scale to production apps. Instead, my technique works by using instrumentation to log the app as it is executed and then creating a visualization of those logs [5]. I implemented this in a tool called AppTracer. I ran AppTracer on 150 popular Android apps to classify the types of interactive resource use in those apps. I used the visualization and a human-curated codebook to assign a set of codes to each permission use.

Next, I conducted a user study to evaluate when users expected permission use to occur. The study is comprised of a vignette interspersed with questions about what resources the user expects will be used. Different participants are assigned different vignettes, in which authorization for resources at different times (on app start, a button click, or not at all).

I found apps largely match user expectations, but there are still key shortcomings with Android's permission system. Users seemed to expect the most invasive permissions (e.g., camera and microphone) to be used only after a click. My app study confirmed this is almost exclusively the case. I 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, Android still asks users for explicit permission via a dialog. This implies Android is being too invasive. More results appear in my CHI 2017 (the top HCI conference) paper.

Permission-Use Provenance in Android

User interfaces bridge implementation and understanding. To exploit this, we need better tools to analyze apps. AppTracer was a first step, but could only detect background permission uses, not *explain* them. The diagram to the right shows a malicious app using the camera in two different ways. Rectangular nodes represent callbacks, red bubbles show permission uses, and edges connect nodes when one handler registers another. Edges are annotated with symbolic formulas detailing facts which hold at registration points.



The use on the far right is benign: a photo is taken after the user clicks a button (invoking the onClick handler). The second is malicious: an intricate sequence of callbacks registers a background thread and waits on a command-and-control server to request a picture be sent. As an example of why symbolic formulas are necessary, ϕ_4 asserts that the command from the server requested a picture be taken. This app would evade the current Android permission system, which authorizes on first use.

[3]This is a weaker property than information flow, but I see this as a first step in understanding human perceptions about interactivity in apps.

Reasoning about permissions-use provenance requires precisely tracking about sequences of callbacks. This is impossible to do at scale with traditional program analyses. My insight was that I could use system logs to circumvent the problems with traditional techniques. This has allowed me to targeted analysis at scale based on recorded app logs.

I implemented this in an analysis tool named Hogarth [3]. Hogarth performs symbolic execution along the paths observed in recorded logs. I applied Hogarth to a set of ten apps and used it to study background permission uses. For example, I found that Bumble—a popular dating app—transmits location in the background upon request from a server to which the app is connected. My work on Hogarth is currently under submission at ICSE 2018, the top software engineering conference.

Future Direction: Analysis-Assisted Auditing Considerable work has gone into checking that systems fit a security policy. Much less work helps us take an arbitrary app and decide the policy. This is especially true for greyware: data uses that aren't explicitly malicious, but still unwanted.[4]

AppTracer and Hogarth are my first steps in this direction. Using program analysis to aid auditing is not a new insight, but has traditionally been hindered by the high number of false alarms in those analyses. However, key to my technique is recognizing that we can combine program analysis with runtime information to help increase its precision. I also want to perform visualization to help auditors understand results of tools like Hogarth, and generalize paths observed at runtime to help find potential security holes that weren't in dynamic paths (such as logic bombs).

Future Direction: Static Analysis to Optimize Dynamic Analyses for Security I strongly believe in using dynamic analyses and tracking for security, because if often applies directly to programs as they exist today, without modifying the language or requiring complicated analysis.

There are many promising dynamic techniques for enforcing security properties at runtime. Unfortunately, these techniques are rarely used in practice because they introduce unacceptable performance overhead. To solve this, I will use program analysis to optimize runtime security checks.

I am currently working on optimizing faceted execution, a runtime monitor for information flow. Faceted execution lifts program variables to include "faceted" variables which track a view of the value as it appears to a privileged party and as it appears to everyone else. Unfortunately, every function in the program must check if the value its working with is faceted, which is quite slow.

This work is hard to perform with students because of it's fairly technical nature. I am collaborating with Thomas Gilray (Maryland) and David Darais (Vermont) to use program analysis to determine how these programs can be optimized. We plan on submitting this work to conferences in early 2018. Once we do that, we plan implement this in practical languages, and this may be a rich opportunity for undergraduate collaboration.

Future Direction: Cross-app security policies So far my work has largely been confined to apps. Now I am stepping out and focusing on what we can do when a user's data is spread across many different devices and services. All large platforms now include privacy controls to help users understand and control how their data may be used. But none of these controls work *together*.

I have begun to push in this direction with three of my undergraduate advisees. We are using dynamic analysis to measure how social-media permissions are used in apps. For example, we want to know if any apps request data from Facebook instead to circumvent the Android permissions system.

To tackle user understanding, I will run user studies to understand how users reason about the composition of different privacy systems being used together. This will happen through a continuing collaboration between myself, Michelle Mazurek and Jeff Foster at Maryland.

[4]For example, apps that record user conversations to make targeted recommendations.

References

[1] Jinseong Jeon, Kristopher Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 3–14, Raleigh, NC, USA, October 2012.

[2] Kristopher Micinski, Jonathan Fetter-Degges, Jinseong Jeon, Jeffrey S. Foster, and Michael R. Clarkson. Checking Interaction-Based Declassification Policies for Android Using Symbolic Execution. In *European Symposium on Research in Computer Security (ESORICS)*, volume 9327 of *Lecture Notes in Computer Science*, pages 520–538, Vienna, Austria, September 2015.

[3] Kristopher Micinski, Thomas Gilray, Daniel Votipka, Jeffrey S. Foster, and Michelle L. Mazurek. Permission-use provenance in android using sparse dynamic analysis, August 2017. In submission to ICSE 2018. Preprint on webpage at `http://kmicinski.com/assets/hogarth.pdf`.

[4] Kristopher Micinski, Philip Phelps, and Jeffrey S. Foster. An Empirical Study of Location Truncation on Android. In *Mobile Security Technologies (MoST)*, San Francisco, CA, May 2013.

[5] Kristopher Micinski, Daniel Votipka, Rock Stevens, Nikolaos Kofinas, Jeffrey S. Foster, and Michelle L. Mazurek. User Interactions and Permission Use on Android. In *Conference on Human Factors in Computing Systems (CHI)*, 2017.