RESEARCH STATEMENT

Kristopher Micinski (kris@cs.haverford.edu)

To be 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 it 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 and formally defined security policies. I used this insight to define a new class of policies that leverages the app's GUI to enforce informed consent. I built tools to measure apps based on these properties and studied users to confirm this fit their mental model.

After starting at Haverford, I wanted to apply the insights from my dissertation to understand why apps used permissions in the background. This required a specific type of reasoning about the program that has proven challenging to tackle with traditional program analyses. My most recent project, Hogarth, uses a novel combination of symbolic execution and dynamic logs to explain in terms of program source why permissions are used in Android apps [6]. This approach has proven highly scalable, and I have used it to study background permission usage in popular apps such as Slack and Bumble. This work was undertaken after I started at Haverford and is currently under review at ICSE 2018, the top software engineering conference.

I have also broadened my research in other directions, and have several active projects in security, programming languages, and HCI. Several of these projects are joint with my undergraduate advisees.

Dynamic Analysis for Security 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].

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 actively maintained and used by a variety of groups in industry and academia. To date the Dr. Android paper has been cited 190 times (240 including citations to 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 parts of the user's location before giving it to apps. We measured this tool on a set of apps and published our results at MOST 13 [7],

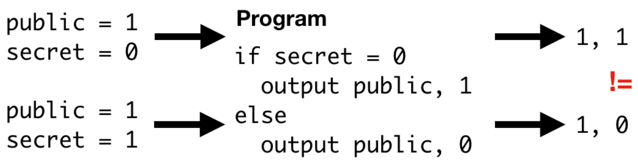
[1] Dr. Android is the Dalvik Rewriter for Android (Dalvik being the bytecode language of Android) and Mr. Hide is the Hide interface to the dalvik environment

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

a workshop in computer security.

Currently, I am with a set of three of my undergraduate advisees in an effort to use dynamic analysis to understand how social-media permissions are used in apps. I believe this is an underexplored direction, and am planning to submit a workshop paper on this topic join with these advisees next semester.

Information Flow and declassification The program below is given a public input and a secret input:



```
if secret = 0
  output public, 1
else
  output public, 0
```

It leaks whether or not it its secret input is zero. To ensure secrecy, it had better be the case that for every fixed input *public*, any pair of *secret* inputs will lead to the same output. This is a violation of *information flow*, a long-studied framework that allows formally defining when a program leaks information. The above program does not directly leak the secret to the output, but ferrys it through the program's control flow. These so-called *implicit flows* have made it challenging to apply traditional techniques such as pointer analysis to information flow. I I began my work in information flow by formalizing a new class of temporal logics—HyperLTL and HyperCTL—that allowed stating information flow properties that involved temporal logics [1]. This work appeared at POST '14.

I next attempted to apply information flow policies to Android apps. I quickly realized that almost all real Android apps violate noninterference to some degree. Because of this problem, various researchers have looked at how to check information flow in the presence of declassification. This merely shifts the problem, because for complex programs (such as Android) it would be unclear what the declassification policy should be. Previous efforts forced programmers to provide annotations. But this would be an unrealistic expectation for the typical app developer. Further, it doesn't help users understand how the policy works.

My insight is 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 as follows: "Assume the phone number was read at some time i, and at some *later* time j, the "send" button is pressed. The phone number can be sent to the network at any time after j, assuming the checkbox for "send phone number" was checked at time j."

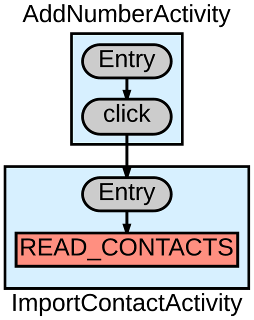
These policies rely on precise temporal orderings of program states. I used *symbolic execution*, a program analysis that executes the program with symbolic variables. In a previous collaboration I worked on SymDroid [3], a symbolic executor for Dalvik bytecode. I extended SymDroid to create ClickRelease. 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. My work on ClickRelease appeared at ESORICS 2015, a strong security conference with a 20% acceptance rate.

Active Project: Abstract Interpretation of Faceted Execution Static analysis for information flow is challenging, especially in the presence of declassification. Dynamic techniques are promising because they can retrofit existing programs without modifying the language or requiring complicated analysis. Faceted execution is one such dynamic technique. In a faceted execution semantics, program variables are divided into normal values and secret (faceted) values of the form $\langle Bob : v_{sec} ? v_{pub} \rangle$, which specify how the value should appear to Bob (v_{sec}) and to everyone except Bob (v_{pub}).

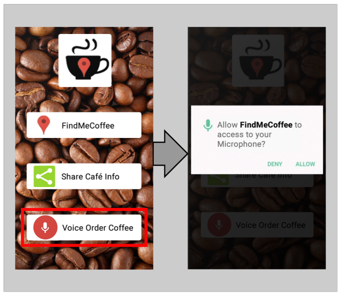
Using faceted execution requires that all functions in the program that will potentially be run on faceted values must be "lifted" to work with these faceted values. This unfortunately means that the program must check at runtime whether each value is faceted or not, significantly impacting performance. I am using a kind of program analysis (abstract interpretation) to eliminate these runtime checks. This is exciting because it will allow immediate performance benefits to any program using faceted execution. Beyond analysis, our technique can more broadly be viewed as a systematic method of applying program analysis to information flow. We plan on submitting this work to conferences in early 2018.

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. So next, I studied how user interaction informs user expectations about sensitive resource usage. I did 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 extensive use of complex libraries. Instead, my technique works by using instrumentation to logs paths through the app as it is executed [8]. 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. 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*.



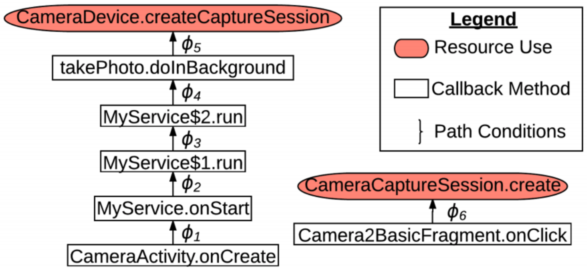
Allow **FindMeCoffee** to access to your Microphone?

Next, I conducted a user study to evaluated 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. 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.

Combining my results, 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 screen. This implies Android is being too invasive. More results appear in my paper which appeared at CHI 2017, the top HCI conference.

Active Project: Permission-Use Provenance in Android

As I applied AppTracer to production apps, I realized that it had problems identifying *why* permissions are used in the background. I wanted to build better tools to help auditors analyze malware. 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 path conditions: symbolic formulas asserting which facts hold when the handler is registered.



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. To see why path conditions are useful, the condition 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 permission use on first use.

Reasoning about permissions-use provenance is challenging because it requires path-sensitive reasoning has never been achieved in popular pointer-based analyses of apps. Techniques amenable to path-sensitive reasoning (such as symbolic execution) have proven impossible to scale to production apps (of the kind I analyzed with AppTracer) because of path explosion and the need for impractically large system models.

My insight was that I could use system logs to circumvent both the lack of a system model and path explosion. This has allowed me to perform targeted symbolic execution at scale for the set of logs I record. First I rewrite the app to insert logging instrumentation. Logging every instruction in an app would be too burdensome. Instead, I identify a minimal set of places in an app where I must add logging instrumentation. Next I run the app to record a corpus of logs. Then I feed this set of logs into Hogarth. Hogarth uses symbolic execution to recover a symbolic replay of the app and uses an SMT solver to produce a permissions-use provenance graph. Each node in this graph corresponds to a callback in the app, and edges connect nodes when one handler registers another.

I applied Hogarth to a set of ten apps: five small benchmark apps on the order of thousands of lines of code. To obtain ground truth, a reverse engineer decompiled each app and manually generated a provenance graph. Hogarth successfully recovered the same graph as the reverse engineer. Next, I applied Hogarth to a set of five production apps. 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.

References

[1] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher Micinski, Markus N. Rabe, and César Sánchez. Temporal Logics for Hyperproperties. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust*, volume 8414 of *Lecture Notes in Computer Science*, pages 265–284. Springer Berlin Heidelberg, 2014.

[2] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010.

[3] Jinseong Jeon, Kristopher Micinski, and Jeffrey S. Foster. SymDroid: Symbolic Execution for Dalvik Bytecode. Technical Report CS-TR-5022, Department of Computer Science, University of Maryland, College Park, July 2012.

[4] 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.

[5] 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.

[6] 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 `https://thomas.gilray.org/pdf/permission-use-provenance.pdf`.

[7] 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.

[8] 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.