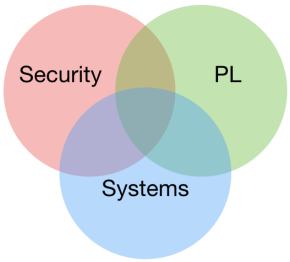
Research Statement

Kristopher Micinski (kris@cs.haverford.edu)

The goal of my research is to help us understand and control potentially-malicious code. To do this, I take hard foundational problems in security and focus on scaling up their solutions to production systems. Thinking about security from an end-to-end perspective means that my work sits at the intersection of three major areas within computer science: security, languages, and systems. Security provides the inspiration for the problems I work on; languages allow me to define and reason formally about security; last, I use techniques from systems to implement my ideas at scale. I strive to set myself apart as a researcher by a will to scale my ideas up to production, and working at the intersection of these areas while building practically-useful systems provides a continual source of exciting and relevant problems.

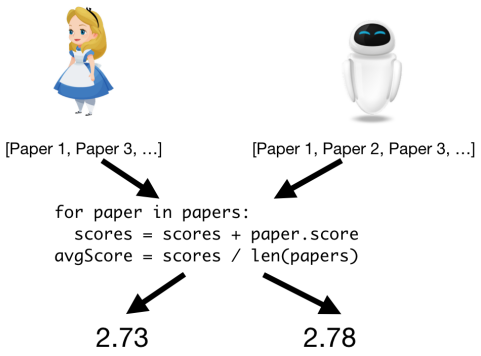


During my PhD I built a system to retrofit Android apps with user-specified security policies, circumventing the policies implemented by the app. Working on this problem, I came up with a hypothesis: well-behaving apps obtain imformed consent via their user-interface. I built tools that helped me gather logs from 150 of the most popular apps—ensuring most well-meaning apps did behave this way—and measured my hypothesis against humans, revealing foundational problems with Android's permission system. This work appeared at top venues (including CHI, the top HCI conference) and my paper on retrofitting Android apps with new policies currently has 238 citations [1]. I have continued to work in that direction, writing a paper that appeared at SOUPS 2018, a strong usable security conference with a 21% acceptance rate[5].

Since then, I have been motivated by two broad problems. The first is building programming languages that allow programmers to enforce declarative security policies without having to bake in the code to implement these policies in application logic. This is an extremely important problem because our security policies are changing more and more frequently (e.g., GDPR). It is no longer sufficient to verify a program meets a static security policy that is assumed fixed for all time. Part of this work was published and presented at the Scheme workshop this September by my undergraduate advisee. The more foundational part of that work was recently submitted to ESOP 2019[2]. The second direction is building better tools for security auditors to vet malware, integrating techniques from program analysis. I designed a novel program analysis methodology that integrates aspects of static and dynamic analysis to help auditors understand why permissions are used. That work was submitted to ICST 2019, the top software testing conference [3].

Fast, Usable Policy-Agnostic Programming

We need the next generation of developers to be using languages that put security front and center. We now know to avoid languages that do not guarantee memory safety, but our current programming languages still have a major issue: they force us to co-mingle security policies and application logic. This means that our systems are very brittle to policy evolution. To make matters worse—while crashes are easy to observe—information leaks are much harder to detect.



To motivate our setting, consider a paper reviewing platform. The system will list all of the papers, along with average scores. Alice (a strong and devious security researcher) is on the PC for this conference, and can see the reviews of all papers except for her own. Luckily, she has discovered a bug: she can calculate the average score of all of the papers she can view and then compare that average to the score she sees (of all of the papers) to figure out her score. Eve—the PC chair—should be able to see all of the papers, and thus the true average score. Of course, once the results are announced, Alice should see the same thing as Eve.

We could ensure this in multiple ways. For example, we might try manually adding in an if statement to the function, filtering the papers so that only the ones Alice was allowed to see were summed. But that would be brittle: as we changed the policy (for example, to allow Alice to see the scores after they are announced) we would need to update the code. If we had to do this for a multi-million line codebase (e.g., as the result GDPR), it would be hard to have any confidence we had done it right.

Optimizing Policy-Agnostic Programming My major contribution is a novel program analysis methodology for *faceted execution*, an implementation strategy for policy-agnostic programming. Faceted execution works by representing secrets as decision trees that specify how the secret should be seen by different observers.

For example, in our system above, faceted execution represents the list of papers as a *facet*: $\langle Eve\ ?\ [p_1,\ p_2,\ p_3, ...] \diamond [p_1, p_3, ...]\rangle$. This can be read as "If Eve views the data, she should see the list $[p_1, p_3, ...]$, otherwise the viewer (Alice) should see $[p_1, p_3, ...]$" (codifying that Alice should not see her own paper). Secret data is marked with these policies at the point that it is read into the system using a special form provided by the language.

When code interacts with a facet (i.e., secret data), it runs both views of the computation: one view as it should appear to Eve and one as it should appear to Alice. In the above example, we can see how faceted execution runs the computation twice, producing two results 2.73 (for Alice's view) and 2.78 (for Eve's view). This result is also encoded as a facet, and for data to leave the system the programmer must explicitly *observe* it specifying the observer. The language runtime then selects the right view of the data to use.

Implementations of faceted execution must consider the fact that whenever the program *possibly encounters* a facet, computation may have to split. This is slow: a compiler for faceted execution would have to add a check for every function call, primitive operation, branch, and so on to handle facets correctly. My insight was that I could use *abstract interpretation* to approximate precisely what values in the program *could be* faceted.

My ESOP 2019 paper outlines a new abstract interpretation strategy for programs that use faceted execution. A key problem arises in designing this analysis: security policies from the program may be arbitrary (first-class) functions, meaning that to reason about facets with any precision at all requires approximating the equivalence of two functions. I solved this by leveraging a trick from abstract interpretation and was able to provide the first sound and precise abstraction for faceted execution. To me, this represents my best work: inspiration from a practical problem that reveals underlying core issues whose solutions lead to broad insights. Now, I want to leverage these foundations to build the next generation of policy-agnostic programming languages.

The Future: Policy-Agnostic Programming at Scale The future I envision is one where we can use the programming languages we use every day, but augment them with policy-agnostic programming. I have begun this by implementing (alongside my undergraduate advisee Zhanpeng Wang) a language called Racets, an extension to the Racket language. While other implementations of faceted execution have used custom runtime systems (which I see as unmaintable long-term), Racets sets itself apart by being built within the core facilities of language (hygienic macros). We (along with Thomas Gilray at Alabama) submitted this work to the Scheme workshop 2018—where it got two strong accepts—and Zhanpeng presented our work this September.

Looking forward, I also want to integrate policy-agnostic programming into more mainstream languages such as Java. My motivation to work on optimizing faceted execution came from trying to implement it as a binary rewriting pass over Java bytecode. I realized the result would be extremely slow. Next, I hope to scale my work from my ESOP paper up to the point that it can be used in real compilers for mainstream languages. Doing so has already revealed other exciting problems. For example, Similarly, our Racets implementation has revealed core issues in interfacing faceted code to plain (unfaceted code). I plan to tackle these issues as I continue to scale fast, usable policy-agnostic programming.

Interaction-Based Security

I have done extensive work on *interaction-based security*, the notion that humans reason about how apps are using their secret data. My work in this area was initially motivated by my work on Android security broadly. I observed that well-behaving apps use the GUI to implement informed consent. I wanted to test this, and to do so I ran two studies to measure how the UI informs expectation of sensitive resource usage.

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 [4]. 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 [4].

In my most recent collaboration, I have studied how users conceptualize *background* data usage. This was the subject of my SOUPS 2018 paper, where we performed studies showing that users' comfort with background data access was highly nuanced, and largely depended on where their data *went*. This was highly motivating to me personally, as it gave a strong indication that the information flow policies from my other project will benefit end-users. This work occurred through a collaboration while I was at Haverford, and appeared at SOUPS 2018, a competitive security venue with a 21% acceptance rate.

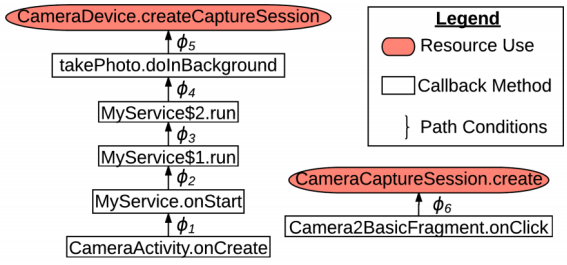
Current Work: Social-Media Permission Usage While Android's systems permissions have been studied extensively by security researchers, there are many more interesting permissions used by apps that I fear might be flying under the radar. The most prominent of these is social-media permission usage. I began to study this with an undergraduate thesis advisee last year, extending my Android logging infrastructure to measure social media permission usage. I plan to continue in this direction, maintaining an actively collaboration with Michelle Mazurek (Maryland) and Jeff Foster (Tufts).

Log-Directed Symbolic Execution

Symbolic Execution (SE) is a highly popular program analysis methodology that runs the program with "symbolic" inputs. For example, consider the program on the right, which makes three API calls and then accesses the user's camera under a set of conditions. Symbolic execution runs the program with each variable replaced by a *symbolic name*. In a concrete execution, the program would decide which branch to take based on the concrete contents of variables. Symbolic execution considers both possible branches, tracking the branch taken by recording a *path condition* (shown in green) as a logical formula. Infeasible branches are typically pruned by calling out to a constraint solver.

```
int x = api_x(); x = α
int y = api_y(); y = β
int z = api_z(); z = γ
if (x < y && y > z) {
  φ = α < β ∧ β > γ
  useCamera();
}
```

My goal is to scale symbolic execution so that it can be readily applied to production Android applications on the order of millions of lines. This has proven intensely challenging because of a few issues. The first is state-space explosion: the farther you symbolically-execute, the more states you have to explore. This is especially pervasive for constructs such as loops over collections, which are used extensively in production code. The second big problem is the framework. Android includes several hundred thousand methods in its standard API. While many symbolic execution research projects formalize a subset of these APIs, none of them can tractably solve this problem, crippling the usability of symbolic execution for production code.



My solution was to cheat. I observed that when reverse engineers perform SE, they first run the app to discover important code before applying SE in a targeted way. So I codified this process into a novel symbolic execution methodology: first log the program's execution, then perform symbolic execution along that log. This sidesteps both the state space explosion and framework modeling problem. The state space is targeted: we only symbolically execute along the log. Similarly, we don't need a framework model: all of the relevant information is encoded into the logs we gather.

I implemented this approach in Hogarth, a symbolic executor that records logs of app runs and uses those logs to explain why an app uses a permission. Hogarth's goal is to drastically reduce the amount of time a security auditor has to spend understanding why an app uses a permission. To use Hogarth, an auditor records a log of an app, feeds this log to Hogarth, and Hogarth produces a *provenance diagram*: a chain of relevant callbacks from an app that caused a permission use.

An example provenance diagram generated by Hogarth is shown on the left (next page). This diagram shows two different uses of the phone's camera. Each node the diagram represents a callback within the app. Edges connect callbacks when one callback registers another. Critically, edges also list path conditions under which the subsequent callback is registered. The use on the right is benign: the user clicks a "take photo" button and the app takes a picture. In this case, ϕ_6 checks that several references are non-null. However—the use on the left is malicious: when the app starts it boots up MyService, which contains a piece of injected malware. This malware then uses a thread to wait for a command from a server to take a picture, at which point it launches a background task to take a picture without the user's authorization. This is encoded in ϕ_4, which states that the data read from the server matched a regular expression including the take_photo string.

Hogarth's implementation relies upon a novel log-collection framework I built, which was carefully engineered to record logs for arbitrarily-complex apps in real time. Hogarth also uses several tricks to perform symbolic execution of the logs efficiently: effectively recovering the information that the logs leave out and giving the auditor a precise reason why the permission use occurred. As a result, I was able to scale Hogarth to twelve of the top 200 apps on Google Play, and Hogarth was effective in understanding why those apps used permissions—analyzing these apps in just minutes. Even more promising, we observed that the time taken to perform an audit using Hogarth's output reduced app auditing time by orders of magnitude. This work was recently submitted to ICST 2019, the top software testing conference. We expect a response December 18th. My colleague and I were then approached by the DOD to submit a proposal for funding to continue that work, and hear back this winter.

Current Work: Speculative Log-Guided Symbolic Execution To me, Hogarth represents a significant achievement in scaling symbolic execution: using ground-truth from real app runs to sidestep both the framework problem and state-space explosion problem. But Hogarth is only a fist step. Because Hogarth simply replays app logs, it cannot uncover *new* bugs in apps, making it useless for (e.g.,) zero-day vulnerability discovery.

I strongly believe a log-directed approach would be highly effective in symbolic execution more broadly, and so I have begun to think about how Hogarth's approach could be adapted to uncover *new* app behavior. In future work, I plan to build a symbolic executor based on Hogarth's log-guided technique that uses the log to guide a search for *new* program behavior. Much of the practically-significant work in symbolic-execution has gone into heuristic search strategies, but very little of that work uses ground-truth from actual program runs to help explore the search space. My goal is to take the results of runs from Hogarth and use them to come up with rich heuristic models of complex APIs, typical of those in large Java and C++ programs. I could imagine this happening in several ways, e.g., using program synthesis to synthesize framework models, or machine learning to understand which parts of the space are more likely to be taken based on the observed logs.

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, David Darais, and Thomas Gilray. Abstracting faceted execution: Static analysis of dynamic information-flow control for higher-order languages. In *submission to ESOP '19*, 2018.

[3] Kristopher Micinski, Thomas Gilray, Daniel Votipka, Jeffrey S. Foster, and Michelle L. Mazurek. Symbolic path tracing to find android permission-use triggers. In *anonymous subission to ICST '19*, 2018.

[4] 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.

[5] Daniel Votipka, Seth M. Rabin, Kristopher Micinski, Thomas Gilray, Michelle M. Mazurek, and Jeffrey S. Foster. User comfort with android background resource accesses in different contexts. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, Baltimore, MD, 2018. USENIX Association.