Monday		
07:30-09:00	breakfast	
09:00-10:00	introductions	
10:00-10:30	coffee	
10:30-11:10	Xavier Leroy	Why compiler correctness says so little about security properties
11:10-12:10	Deepak Garg	What is secure compilation? A property-centric view
12:15-14:00	lunch	
14:00-15:00	Peter Sewell	Secure Compilation – understanding the endpoints?
15:00-15:30	David Chisnall	Teaching a production compiler that integers are not pointers
15:30-16:00	Magnus Myreen	Is the verified CakeML compiler secure?
16:00-16:30	cake	
16:30-18:00	Lead: Catalin Hritcu	Discussion: Secure Compilation Goals and Attacker Models
Tuesday		
07:30-09:00	breakfast	
09:00-10:00	Amal Ahmed	Compositional compiler correctness and secure compilation: Where we are and where we want to be.
10:00-10:30	coffee	
10:30-10:50	David Chisnall	Preserving high-level invariants in the presence of low-level code
10:50-11:30	Dominique Devriese	Capability machines as a target for secure compilation
11:30-12:10	Akram El-Korashy	A secure compiler from C to CHERI
12:15-13:30	lunch	
13:30-16:00	hike around Dagstuhl	
16:00-16:30	cake	
16:30-18:00	Working in groups (LLVM, Spectre, etc)	
Wednesday		
07:30-09:00	breakfast	
09:00-09:30	Steve Zdancewic	Call-by-Push-Value and Reasoning about Low-Level IRs
09:30-09:50	Christine Rizkallah	A Formal Equational Theory for Call-By-Push-Value
09:50-10:10	Chris Hawblitzel	A Spectre haunts our secure compilers
10:10-10:40	coffee	
10:40-11:10	Deian Stefan	Constant-time crypto programming with FaCT
11:10-11:50	Daniel Patterson	Linking Types: Bringing Fully Abstract Compilers and Flexible Linking Together
11:50-12:10	Nick Benton	Thoughts on preserving abstractions
12:15-14:00	lunch	
14:00-14:40	Pramod Bhatotia	Memory safety for Shielded Execution
14:40-15:10	Santosh Nagarakatte	Compiler Optimizations with Retrofitting Transformations: Is there a Semantic Mismatch?
15:10-15:40	John Criswell	Virtual Instruction Set Computing with Secure Virtual Architecture
15:40-16:00	Max New	Specifications for Dynamic Enforcement of Relational Program Properties
16:00-16:30	cake	
16:30-18:00	Lead: Frank Piessens	Discussion: Effective Enforcement Mechanisms for Secure Compilation
Thursday		
07:30-09:00	breakfast	
09:00-09:30	Derek Dreyer	Defining Undefined Behavior in Rust
09:30-09:35	Dave Naumann	Relational Logic for Fine-grained Security Policy and Translation Validation

09:35-09:40	Frédéric Besson	CompCertSFI: Formally Veried Software Fault Isolation
09:40-09:45	Zoe Paraskevopoulou	Closure Conversion is Safe-for-Space
09:45-09:50	Limin Jia	Taming I/O in Intermittent Computing
10:20-10:50	coffee	
10:50-11:30	Catalin Hritcu	Formally Secure Compilation of Unsafe Low-level Components
11:30-12:00	Andrew Tolmach	C-level tag-based security monitors
12:10-12:15	Group Photo	
12:15-14:00	lunch	
14:00-14:20	Chung-Kil Hur	Taming Undefined Behavior in LLVM
14:20-15:00	Toby Murray	Verified Compilation of Noninterference for Shared-Memory Concurrent Programs
15:00-15:30	Stefan Brunthaler	Software Diversity vs. Side Channels
15:30-16:00	Kedar Namjoshi	Plugging Leaks Introduced by Compiler Optimizations
16:00-16:30	cake	
16:30-18:00	Lead: Amal Ahmed	Discussion: Formal verification and proof techniques
Friday		
07:30-09:00	breakfast	
09:00-09:30	Stephanie Weirich	Verifying the Glasgow Haskell Compiler Core language
09:30-09:50	Gabriele Keller	Data Refinement for Cogent
09:50-10:20	Frédéric Besson	Preservation of safe erasure as an information flow property
10:20-10:50	coffee	
10:50-11:10	Tamara Rezk	A project on secure compilation in the context of the IoT
11:10-11:30	Cédric Fournet	Building Secure SGX Enclaves using F*, C/C++ and X64
11:30-11:50	Vincent Laporte	Secure compilation of side-channel countermeasures: the case of cryptographic “constant-time”
12:15-14:00	lunch	

Participant name	Title	Abstract (can be informal)	Collaborators (especially if also attending)	Duration
Akram El-Korashy	A secure compiler from C to CHERI	Capability machines offer architectural support for fine-grained memory separation and controlled sharing. In this in-progress work, we leverage this support to compile a high-level data isolation primitive fully abstractly. We start from a safe subset of C extended with an abstraction for modules that may have private state. The language semantics prevent a module from accessing an element of another module's private state, unless it has been shared explicitly. We then describe a compiler from this language to CHERI, a modern capability machine. In ongoing work, we are proving that the compiler is fully abstract, i.e., it preserves and reflects observational equivalence and, hence, implements the source module abstraction securely.	Stelios Tsampas, Marco Patrignani, Dominique Devriese, Frank Piessens, Deepak Garg	20 + 20
Amal Ahmed	Compositional compiler correctness and secure compilation: Where we are and where we want to be.	In this talk, I'll start with a brief but insightful survey of recent compositional compiler correctness results. I'll give a high-level perspective on what is good and bad about each of the existing compositional compiler correctness results and how their formalisms influence the required verification effort. I'll explain why _none_ of the compositional compiler correctness results to date are where we want to be! Then I'll present a generic compositional compiler correctness (CCC) theorem that abstracts away from existing formalisms. CCC gives us insight on what is required for modular verification of multi-pass compilers. I will end with an insight for those working on secure compilation results that require "weaker" protection of compiled components than fully abstraction compilation: when it comes to proving such compilers correct, truly modular verification of multi-pass compilers seems impossible.	Daniel Patterson	40 + 20 tutorial (very few slots)
Andrew Tolmach	C-level tag-based security monitors.	Recent work on security "micropolicies" uses hardware-level metadata tags to monitor individual machine operations. This talk will sketch preliminary ideas for how to raise the definition of tag-based policies to the level of C code. C-level polices should be useful both to express high-level properties that are tedious or impossible to specify at machine level (e.g. information flow control or compartmentalization) and to enforce particular variants of C semantics (e.g. differing flavors of memory safety based on differing pointer aliasing rules). C-level policies can be (verifiably) compiled to machine-level policies to be enforced by existing (prototype) hardware.	Catalin Hritcu, Benjamin Pierce, Sean Anderson (student)	15 + 15
Catalin Hritcu	Formally Secure Compilation of Unsafe Low-level Components	We propose a new formal criterion for secure compilation, giving end-to-end security guarantees for software components written in unsafe, low-level languages with C-style undefined behavior. Our criterion is the first to model *dynamic* compromise in a system of mutually distrustful components with clearly specified privileges. Each component is protected from all the others---in particular, from components that have encountered undefined behavior and become compromised. Each component receives secure compilation guarantees up to the point when it becomes compromised, after which an attacker can take complete control over the component and use its privileges to attack the remaining uncompromised components.	Andrew Tolmach (attending), Guglielmo Fachini, Marco Stronati, Arthur Azevedo de Amorim, Ana Nora Evans, Carmine Abate, Roberto Blanco (attending), Théo Laurent, Benjamin C. Pierce.	20 + 20
Cédric Fournet	Building Secure SGX Enclaves using F*, C/C++ and X64	Intel SGX offers hardware mechanisms to isolate code and data running within enclaves from the rest of the platform. This enables security verification on a relatively small software TCB, but the task still involves complex low-level code. Relying on the Everest verification toolchain, we use F* for developing specifications, code, and proofs; and then safely compile F* code to standalone C code. However, this does not account for all code running within the enclave, which also includes trusted C and assembly code for bootstrapping and for core libraries. Besides, we cannot expect all enclave applications to be rewritten in F*, so we also compile legacy C++ defensively, using variants of /guard that dynamically enforce their safety at runtime. To reason about enclave security, we thus compose different sorts of code and verification styles, from fine-grained statically-verified F* to dynamically-monitored C++ and custom SGX instructions. This involves two related program semantics: most of the verification is conducted within F* using the target semantics of Kremlin—a fragment of C with a structured memory—whereas SGX features and dynamic checks embedded by defensive C++ compilers require lower-level X64 code, for which we use the verified assembly language for Everest (VALE) and its embedding in F*.	Anitha Gollamudi	10 + 10
Chris Hawblitzel	A Spectre haunts our secure compilers	Hardware is full of side channels that thwart our attempts to execute software securely. The recent Spectre vulnerability is one of the most worrisome. What is Spectre, and what mitigations against it have been applied to our hardware, applications, and compilers? How can we formally reason about information leakage in the presence of speculation and memory side channels? Given the tradeoffs between performance and side channel freedom, what guarantees would we like hardware to provide to software?		10 + 10
Christine Rizkallah	A Formal Equational Theory for Call-By-Push-Value	Establishing that two programs are contextually equivalent is hard, yet essential for reasoning about semantics preserving program transformations such as compiler optimizations. The Vellvm project aims to use Coq to formalize and reason about LLVM program transformations and as part of this project we are using a variant of Levy's call-by-push-value language. I will talk about how we establish the soundness of an equational theory for call-by-push-value and about how we used our equational theory to significantly simplify the verification of classic optimizations.	Steve Zdancewic	10 + 10
Chung-Kil Hur	Taming Undefined Behavior in L	A central concern for an optimizing compiler is the design of its intermediate representation (IR) for code. The IR should make it easy to perform transformations, and should also afford efficient and precise static analysis. In this paper we study an aspect of IR design that has received little attention: the role of undefined behavior. The IR for every optimizing compiler we have looked at, including GCC, LLVM, Intel's, and Microsoft's, supports one or more forms of undefined behavior (UB), not only to reflect the semantics of UB-heavy programming languages such as C and C++, but also to model inherently unsafe low-level operations such as memory stores and to avoid over-constraining IR semantics to the point that desirable transformations become illegal. The current semantics of LLVM's IR fails to justify some cases of loop unswitching, global value numbering, and other important "textbook" optimizations, causing long-standing bugs. We present solutions to the problems we have identified in LLVM's IR and show that most optimizations currently in LLVM remain sound, and that some desirable new transformations become permissible. Our solutions do not degrade compile time or performance of generated code.		10 + 10
Daniel Patterson	Linking Types: Bringing Fully Abstract Compilers and Flexible Linking Together	Fully abstract compilers protect components from target-level attackers by ensuring that any observations or influence that a target attacker could have can also be done by a source-level attacker. This means that programmers need only reason about security properties in their own language, not additional interactions that may happen in lower level intermediate or target languages. While this is obviously an extremely valuable property for secure compilers, it rules out linking with target code that has features or restrictions that can not be represented in the source language that is being compiled. While traditionally fully abstract compilation and flexible linking have been thought to be at odds, I'll present a novel idea called Linking Types that allows them to coexist. Linking Types enable a programmer to opt in to local violations of full abstraction that she needs in order to link with particular code without giving up the property globally. This fine-grained mechanism enables flexible interoperation with low-level features while preserving the high-level reasoning principles that fully abstract compilation offers. The talk will give some brief background to the ideas, show how they play out in examples, and open a broader discussion as to how this idea could influence secure compilers and language design.	Amal Ahmed	20 + 20
Dave Naumann	Relational Logic for Fine-grained Security Policy and Translation Validation	Relational Hoare logics facilitate reasoning about information-flow properties of programs as well as relations between programs such as observational equivalence. Such logics might be used to specify sensitive information at source level and to specify what is considered observable at source and target levels, in order to define security-preserving compilation and support translation validation. In this 5-10 min talk I could sketch these ideas and get feedback on how they could be investigated further.		5 minutes
David Chisnall	Preserving high-level invariants in the presence of low-level code	Most complex programs contain a mixture of different language, but the guarantees available in common implementations are those of the lowest-level language. A typical Java implementation includes well over a million lines of C/C++ code with no constraints on its abilities and the same is true for most other high-level languages. In the CHERI JNI work presented at ASPLOS last year, we demonstrated one possible way of allowing untrusted native code (including unverified assembly code) to exist in the same process as Java code, with high performance and preserving all of the invariants on which the Java security model is built.	Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannouand Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie and Robert N. M. Watson	10 + 10

Participant name	Title	Abstract (can be informal)	Collaborators (especially if also attending)	Duration
David Chisnall	**Teaching a production compiler that integers are not pointers**	Over the past six years, we have created taught the clang front end for [Objective-]C/C++, the LLVM optimisation pipeline, and the MIPS back end, to understand that pointers are a distinct type from integers (though memory may contain either). With the CHERI extensions applied to MIPS, we are able to preserve the distinction between pointers and integers all of the way from a source language, which supports features such as untagged unions and untyped memory, all of the way through the compilation pipeline to hardware that can preserve this distinction at run time. We support a single-provenance semantics for pointers and can discuss the changes required to the compiler and our design decisions for concrete choices allowed within the C/C++ abstract machine that maintain compatibility with large corpora of real-world code while preserving memory safety.	Khilan Gudka, Alex Richardson, Peter Sewell	15 + 15
Deepak Garg	**What is secure compilation?**	What does it mean that a compiler chain is secure? How does one define such secure compilation formally? And to what attacker model does it correspond? In this talk I will argue that a secure compilation chain should preserve some well-specified class of security properties of source programs even against adversarial low-level contexts. Particularly interesting classes include safety properties, hyperproperties (e.g. non-interference), and relational hyperproperties (e.g. observational equivalence).	Catalin Hritcu (attending), Marco Patrignani (also attending), Carmine Abate, Jérémy Thibault	40 + 20 tutorial (very few slots)
Deian Stefan	**Constant-time crypto programming with FaCT**	Implementing cryptographic algorithms that do not inadvertently leak secret information is notoriously difficult. Today's general-purpose programming languages and compilers do not account for data sensitivity; consequently, most real-world crypto code is written in a subset of C intended to predictably run in constant time. This C subset, however, forgoes structured programming as we know it -- crypto developers, today, do not have the luxury of if-statements, efficient looping constructs, or procedural abstractions when handling sensitive data. Unsurprisingly, even high-profile libraries, such as OpenSSL, have repeatedly suffered from bugs in such code. In this talk, I will describe FaCT, a new domain-specific language that addresses the challenge of writing constant-time crypto code. With FaCT, developers write crypto code using standard, high-level language constructs; FaCT, in turn, compiles such high-level code into constant-time assembly. FaCT is not a standalone language. Rather, we designed FaCT to be embedded into existing, large projects and language. In this talk, I will describe how we integrated FaCT in several such projects (OpenSSL, libsodium, and mbedtls) and languages (C, Python, and Haskell).		15 + 15
Derek Dreyer	**Defining Undefined Behavior in Rust**	In the RustBelt project, we have been building foundations for understanding the safety claims of the Rust language and for evolving the language safely. In so doing, we have thus far assumed a memory model in which the only forms of undefined behavior are data races and memory safety violations. However, this is too simplistic. The Rust developers would like to support more aggressive compiler optimizations that exploit non-aliasing assumptions derived from Rust's reference types, but in order for such optimizations to be sound, undefined behavior must be expanded to include unsafe code that violates such non-aliasing assumptions. In this talk, I will report on several avenues currently being explored for defining undefined behavior in Rust. I can give either a 10-minute talk or a 15-minute talk, depending on how much detail people want to hear. This is very much work in progress.	Ralf Jung	15 + 15
Dominique Devriese	**Capability machines as a target for secure compilation**	A quick introduction to capability machines, and an overview of ideas about how different properties can be enforced using different extensions of capability machines	Thomas Van Strydonck (not attending), Frank Piessens, Lau Skorstengaard (not attending), Lars Birkedal, Akram El-Korashy, Stelios Tsampas (not attending), Marco Patrignani, Deepak Garg	20 + 20
Frédéric Besson	**Preservation of safe erasure as an information flow property**	Secure coding requires erasing secrets to limit the possibility for an attacker to probe the content of memory. At source level, erasure is typically performed by a memset (secret,0). Yet, as secret is dead, compiler optimisations may remove this piece of code and therefore break the security. In the talk, I will test on the audience a semantics definition of (preservation) of safe erasure phrased in terms of quantitative information flow. I will then sketch how typical compiler optimisations (DSE, register allocation) need to be modified to preserve this property.		15 + 15
Frédéric Besson	**CompCertSFI**	Formally Veried Software Fault Isolation		5 minutes
Gabriele Keller	**Data Refinement for Cogent**	COGENT allows low-level operating system components to be modelled as pure mathematical functions operating on algebraic data types, suitable for verification in an interactive theorem prover. Further-more, it can compile these models into imperative C programs, and provide a proof that this compilation is a refinement of the functional model. Currently, however, there is still a gap between the C data structures used in the operating system, and the algebraic data types used by COGENT, which force the programmer to write a large amount of boilerplate marshalling code to connect the two. In this talk, I'll outline our current work on adding a data description component to the framework, which will allow COGENT to be flexible in how it represents its algebraic data types, enabling models that operate on standard algebraic data types to be compiled into C programs that manipulate C data structures directly. Once fully realised, this extension will enable more code to be automatically verified by COGENT, smoother interoperability with C, and substantially improved performance of the generated code.	Christine Rizkallah	10+10
John Criswell	**Virtual Instruction Set Computing with Secure Virtual Architecture**	This talk will present Secure Virtual Architecture (SVA): a virtual instruction set computing infrastructure which we have used to enforce security policies on both application and operating system kernel code. I will present how we have used SVA to enforce traditional policies like memory safety and control flow integrity as well as newer policies such as newer policies that mitigate side-channel attacks and Spectre/Meltdown attacks launched by compromised operating system kernels. I hope to solicit feedback on how to employ secure compilation techniques into SVA to further reduce its (already small) trusted computing base size and to discuss the use of secure compilation techniques on operating system kernel code.		15 + 15
Kedar Namjoshi	**Plugging Leaks Introduced by Compiler Optimizations**	Some compiler optimizations (e.g., dead store removal, or SSA conversion) can introduce new information leaks as they transform a program. I will talk about sound -- but necessarily approximate -- methods to produce leak-free forms of these optimizations. Not all optimizations introduce leaks; I will show how one can verify that an implementation of a transformation is leak-free by checking additional properties of a refinement relation (a "witness") that is produced originally to justify correctness. There are several open questions (e.g., how to establish preservation of security properties other than information leakage?) which I hope to have the chance to discuss during the talk and in the seminar.		15 + 15
Limin Jia	**Taming I/O in Intermittent Computing**	Energy harvesting enables novel devices and applications without batteries. However, intermittent operation under energy harvesting poses new challenges to preserving program semantics under power failures. I will first discuss uniques challenges that existing check-pointing mechanisms for intermittent computing face in the presence of I/O operations. Then, I will talk about our ongoing work on developing a static analysis tool for automatically identifying bugs caused by I/O operations, methods for fixing such bugs, and formal models for intermittent computing.		5 minutes
Magnus Myreen	**Is the verified CakeML compiler secure?**	I propose to (1) present the CakeML compiler at a high-level, then (2) zoom in on the exact details of the compiler correctness theorem, but leave plenty of time for (3) a discussion on whether the CakeML compiler is secure or not. The CakeML compiler starts from a safe language (unsafe out-of-bounds accesses are not possible) and compiles it to concrete machine code (x86, ARM, RISC-V etc.) with a semantics where the OS and other programs are allowed to interrupt the CakeML machine code. The CakeML compiler is probably safer than unverified compilers for ML, but is it more secure? In the discussion part of my talk, I'll talk about different attacker models and security questions regarding the target semantics which is at the level of machine code. I would ideally like to talk for 10-15 minutes and have 15-20 minutes for discussion.	Scott Owens (attending), Ramana Kumar, Michael Norrish, Yong Kiam Tan, Anthony Fox	15 + 15

Participant name	Title	Abstract (can be informal)	Collaborators (especially if also attending)	Duration
Max New	**Specifications for Dynamic Enforcement of Relational Program Properties**	Many security and reliability properties are phrased in terms of relations on programs, e.g., noninterference and representation independence. While all source-level programs respect these relational properties due to syntactic restrictions such as linearity or type checking, when compiling securely to low-level programs, we need to interpose on the boundary between compiled code and low-level attackers to maintain our high-level security properties. In this talk we present a simple specification for the interposition functions between compiled code and low-level attackers. The basic idea is to first provide a *refinement relation* between high level and low level behaviors. Some simple properties must be satisfied to ensure that the refinement relation is compatible with the relational properties of interest. Then functions that enforce high-level interfaces on low-level attackers and dually protect compiled code from low-level attackers can be given two dual specifications with respect to the refinement relation. An enforcement function is sound if its output refines its input, and *optimal* if it has the most behavior of any refinement of the input. Dually, a protection function is sound if its output is refined by its input, and *optimal* if it has the least behavior of any refinement of the input. Finally, to get security/full abstraction we need the protection function to be *injective*, which is here equivalent to saying that `enforce o protect = id`. This fairly simple spec is the core of "galois connection"-based approaches to security, but we argue that by focusing on the refinement relation first, the galois connection properties become more intuitive. Furthermore, since the actual implementation of enforce and protect can be quite complex, it is useful to specify them first in terms of a simple refinement relation.	Amal Ahmed	10 + 10
Nick Benton	**Thoughts on preserving abstractions**			10 + 10
Peter Sewell	**Secure Compilation – understanding the endpoints?**	Short update for several related projects under our REMS umbrella focusing on the bits most relevant to secure compilation: a) our Sail-based work on ISA semantics, towards more-or-less complete sequential ISA specs for ARMv8-A (derived from the ARM-internal specification), CHERI, and RISC-V, with smaller IBM POWER and x86 fragments. We aim to produce usable Isabelle and Coq versions for others to build on. b) hardware concurrency semantics, mostly for ARM and RISC-V c) proving security properties of CHERI d) sequential C source semantics and WG14 - and its relation to CHERI C e) WebAssembly semantics	(a) Alasdair Armstrong, Thomas Bauereiss, Brian Campbell (Edinburgh), Shaked Flur, Kathryn E. Gray (now Facebook), Neel Krishnaswami, Prashanth Mundkur (SRI), Robert M. Norton, Christopher Pulte, Alastair Reid (ARM), Ian Stark (Edinburgh), Mark Wassell (b) Shaked Flur, Christopher Pulte, Gil Hur (SNU), Jean Pichon-Pharabod, Luc Maranget (INRIA), Susmit Sarkar (St Andrews) (c) Kyndylan Nienhuis and the CHERI team (d) Kayvan Memarian, Victor B. F. Gomes (e) Conrad Watt	40 + 20 tutorial (very few slots)
Pramod Bhatotia	**Memory safety for Shielded Execution**	In this talk, I will first present our work on SGXBounds on how to achieve lightweight memory safety in the context of SGX Enclaves. http://se.inf.tu-dresden.de/pubs/papers/sgxbounds2017.pdf I will conclude the talk with our on-going work on Intel MPX Explained: https://intel-mpx.github.io/		20 + 20
Santosh Nagarakatte	**Compiler Optimizations with Retrofitting Transformations: Is there a Semantic Mismatch?**	A retrofitting transformation modifies an input program by adding instrumentation to monitor security properties at runtime. These tools often transform the input program in complex ways. Compiler optimizations can erroneously remove the instrumentation added by a retrofitting transformation in the presence of semantic mismatches between the assumptions of retrofitting transformations and compiler optimizations. This talk will describe a generic strategy to ascertain that every event of interest that is checked in the retrofitted program is also checked after optimizations.		15 + 15
Stefan Brunthaler	**Software Diversity vs. Side Channels**	The past couple of years have seen attacks becoming increasingly sophisticated, primarily due to the discovery and incorporation of side channels. For example, Drammer, AnC, and SPECTRE showed how predictable behavior enables modern side-channel attacks. Based on my experience with using diversity to counter timing-based side-channel attacks (cf. NDSS'15 paper), I have devised a couple of new diversity defenses to thwart Drammer and substantially lessen the impact of SPECTRE attacks.	n/a	15 + 15
Stephanie Weirich	**Verifying the Glasgow Haskell Compiler Core language**	Verified compilers are one part of secure compilation. By developing a compiler within the language of a proof assistant, we can rigorously show that the semantics of the source language is preserved through compilation to the target. However, what about our existing compilers? In this talk, I will present our preliminary work that uses the Coq theorem prover to reason about the implementation of the GHC Core intermediate language. Our goal is to show that Core optimization passes are correct: i.e. that these transformations preserve the invariants of the compiler AST and, ultimately, the semantics of the Core language. Our work uses the hs-to-coq tool to translate the source code of GHC from Haskell into Gallina, the language of the Coq proof assistant, taking advantage of the similarity between the languages. One discussion point is how much our proofs actually apply to GHC --- what can we really prove about compilation and what guarantees can we conclude from our work?	Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley	15 + 15
Steve Zdancewic	**Call-by-Push-Value and Reasoning about Low-Level IRs**	Real-world compilers use control-flow-graph-based intermediate representations. For example, the LLVM IR consists of control-flow-graphs structured according to the static single assignment (SSA) invariants. Such IRs are well-suited for backend code generation and implementing analyses and optimization passes; however, formalizing such IRs and reasoning about the correctness of those analyses and optimizations at that level can be challenging. In the Vellvm (Verified LLVM) project, we have been experimenting with representing SSA control-flow-graphs using terms of Levy's call-by-push-value (CBPV) variant of the lambda calculus. CBPV offers the benefits of a good equational theory based on the usual notions of beta-equivalence. By relating the operational semantics of the CBPV language to that of the SSA-control-flow graphs, we can transport reasoning and program transformations from one level to another, thereby allowing for very simple proofs of the correctness of many low-level optimizations such as function inlining. This talk will explain our on-going work in this area and conections to the LLVM IR.	Christine Rizkallah (attending -- she will talk about a different, but related piece of this project) Dmitri Garbuzov, William Mansky, and Yannick Zakowski.	15 + 15
Tamara Rezk	**A project on secure compilation in the context of the IoT**	I will briefly present a new starting project which relies on the idea of using secure compilation for the Internet of Things (IoT). The talk will present new challenges in the IoT context, security risks, and speculations on how to address them. http://cisc.gforge.inria.fr/	Frédéric Besson, Thomas Jensen, Alan Schmitt, Gérard Berry, Nataliia Bielova, Ilaria Castellani, Manuel Serrano, Claude Castelluccia, Daniel Le Métayer	10+10
Toby Murray	**Verified Compilation of Noninterference for Shared-Memory Concurrent Programs**	I propose to present our work on verified compilation of (value-dependent) noninterference for concurrent programs. I would present the underlying theory (definitions of secure refinement) and their instantiation in the context of a compiler from a simple While language to an idealised RISC language. I would present the current state of the work, future plans, opportunities for collaboration, relationship to other ongoing work on verified noninterference for concurrent programs, etc.	Christine Rizkallah	20 + 20
Xavier Leroy	**Why compiler correctness says so little about security properties**	I could talk 10-15 minutes on the basics of compiler verification and 10-15 minutes (plus copious discussions, I'm afraid) on why a CompCert-style compiler verification says so little about security properties and what could possibly be done about it, with preservation of constant-time-ness as an example.		20 + 20
Zoe Paraskevopoulou	**Closure Conversion is Safe-for-Space**	Compiler transformations may fail to preserve the resource consumption of compiled programs. A notable example is closure conversion with linked closures which may introduce space leaks. In this talk I will present a (currently ongoing) proof that closure conversion with flat closure representation is safe-for-space, meaning that it preserves the space complexity of the compiled program. We develop a method based on step-indexed logical relations that allows us to conveniently reason about the resource consumption of the source and target programs, as well as the functional correctness of the transformation.	Andrew Appel	5 minutes