<table>
<thead>
<tr>
<th>Monday</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>07:30-09:00</td>
<td>breakfast</td>
<td></td>
</tr>
<tr>
<td>09:00-10:00</td>
<td>introductions</td>
<td></td>
</tr>
<tr>
<td>10:00-10:30</td>
<td>coffee</td>
<td></td>
</tr>
<tr>
<td>10:30-11:10</td>
<td>Xavier Leroy</td>
<td>Why compiler correctness says so little about security properties</td>
</tr>
<tr>
<td>11:10-12:10</td>
<td>Deepak Garg</td>
<td>What is secure compilation? A property-centric view</td>
</tr>
<tr>
<td>12:15-14:00</td>
<td>lunch</td>
<td></td>
</tr>
<tr>
<td>14:00-15:00</td>
<td>Peter Sewell</td>
<td>Secure Compilation – understanding the endpoints?</td>
</tr>
<tr>
<td>15:00-15:30</td>
<td>David Chisnall</td>
<td>Teaching a production compiler that integers are not pointers</td>
</tr>
<tr>
<td>15:30-16:00</td>
<td>Magnus Myreen</td>
<td>Is the verified CakeML compiler secure?</td>
</tr>
<tr>
<td>16:00-16:30</td>
<td>cake</td>
<td></td>
</tr>
<tr>
<td>16:30-18:00</td>
<td>Lead: Catalin Hritcu</td>
<td>Discussion: Secure Compilation Goals and Attacker Models</td>
</tr>
<tr>
<td>Tuesday</td>
<td></td>
<td></td>
</tr>
<tr>
<td>07:30-09:00</td>
<td>breakfast</td>
<td></td>
</tr>
<tr>
<td>09:00-10:00</td>
<td>Amal Ahmed</td>
<td>Compositional compiler correctness and secure compilation: Where we are and where we want to be.</td>
</tr>
<tr>
<td>10:00-10:30</td>
<td>coffee</td>
<td></td>
</tr>
<tr>
<td>10:30-10:50</td>
<td>David Chisnall</td>
<td>Preserving high-level invariants in the presence of low-level code</td>
</tr>
<tr>
<td>10:50-11:30</td>
<td>Dominique Devriese</td>
<td>Capability machines as a target for secure compilation</td>
</tr>
<tr>
<td>11:30-12:10</td>
<td>Akram El-Korashy</td>
<td>A secure compiler from C to CHERI</td>
</tr>
<tr>
<td>12:15-13:30</td>
<td>lunch</td>
<td></td>
</tr>
<tr>
<td>13:30-16:00</td>
<td>hike around Dagstuhl</td>
<td></td>
</tr>
<tr>
<td>16:00-16:30</td>
<td>cake</td>
<td></td>
</tr>
<tr>
<td>16:30-18:00</td>
<td>Working in groups (LLVM, Spectre, etc)</td>
<td></td>
</tr>
<tr>
<td>Wednesday</td>
<td></td>
<td></td>
</tr>
<tr>
<td>07:30-09:00</td>
<td>breakfast</td>
<td></td>
</tr>
<tr>
<td>09:00-09:30</td>
<td>Steve Zdancewic</td>
<td>Call-by-Push-Value and Reasoning about Low-Level IRs</td>
</tr>
<tr>
<td>09:30-09:50</td>
<td>Christine Rizkallah</td>
<td>A Formal Equational Theory for Call-By-Push-Value</td>
</tr>
<tr>
<td>09:50-10:10</td>
<td>Chris Hawblitzel</td>
<td>A Spectre haunts our secure compilers</td>
</tr>
<tr>
<td>10:10-10:40</td>
<td>coffee</td>
<td></td>
</tr>
<tr>
<td>10:40-11:10</td>
<td>Deian Stefan</td>
<td>Constant-time crypto programming with FaCT</td>
</tr>
<tr>
<td>11:10-11:50</td>
<td>Daniel Patterson</td>
<td>Linking Types: Bringing Fully Abstract Compilers and Flexible Linking Together</td>
</tr>
<tr>
<td>11:50-12:10</td>
<td>Nick Benton</td>
<td>Thoughts on preserving abstractions</td>
</tr>
<tr>
<td>12:15-14:00</td>
<td>lunch</td>
<td></td>
</tr>
<tr>
<td>14:00-14:40</td>
<td>Pramod Bhatotia</td>
<td>Memory safety for Shielded Execution</td>
</tr>
<tr>
<td>14:40-15:10</td>
<td>Santosh Nagarakatte</td>
<td>Compiler Optimizations with Retrofitting Transformations: Is there a Semantic Mismatch?</td>
</tr>
<tr>
<td>15:10-15:40</td>
<td>John Criswell</td>
<td>Virtual Instruction Set Computing with Secure Virtual Architecture</td>
</tr>
<tr>
<td>15:40-16:00</td>
<td>Max New</td>
<td>Specifications for Dynamic Enforcement of Relational Program Properties</td>
</tr>
<tr>
<td>16:00-16:30</td>
<td>cake</td>
<td></td>
</tr>
<tr>
<td>16:30-18:00</td>
<td>Lead: Frank Piessens</td>
<td>Discussion: Effective Enforcement Mechanisms for Secure Compilation</td>
</tr>
<tr>
<td>Thursday</td>
<td></td>
<td></td>
</tr>
<tr>
<td>07:30-09:00</td>
<td>breakfast</td>
<td></td>
</tr>
<tr>
<td>09:00-09:30</td>
<td>Derek Dreyer</td>
<td>Defining Undefined Behavior in Rust</td>
</tr>
<tr>
<td>09:30-09:35</td>
<td>Dave Naumann</td>
<td>Relational Logic for Fine-grained Security Policy and Translation Validation</td>
</tr>
<tr>
<td>Time</td>
<td>Speaker</td>
<td>Title</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>09:35-09:40</td>
<td>Frédéric Besson</td>
<td>CompCertSFI: Formally Verified Software Fault Isolation</td>
</tr>
<tr>
<td>09:40-09:45</td>
<td>Zoe Paraskevopoulou</td>
<td>Closure Conversion is Safe-for-Space</td>
</tr>
<tr>
<td>09:45-09:50</td>
<td>Limin Jia</td>
<td>Taming I/O in Intermittent Computing</td>
</tr>
<tr>
<td>10:20-10:50</td>
<td></td>
<td>coffee</td>
</tr>
<tr>
<td>10:50-11:30</td>
<td>Catalin Hritcu</td>
<td>Formally Secure Compilation of Unsafe Low-level Components</td>
</tr>
<tr>
<td>11:30-12:00</td>
<td>Andrew Tolmach</td>
<td>C-level tag-based security monitors</td>
</tr>
<tr>
<td>12:10-12:15</td>
<td></td>
<td>Group Photo</td>
</tr>
<tr>
<td>12:15-14:00</td>
<td></td>
<td>lunch</td>
</tr>
<tr>
<td>14:00-14:20</td>
<td>Chung-Kil Hur</td>
<td>Taming Undefined Behavior in LLVM</td>
</tr>
<tr>
<td>14:20-15:00</td>
<td>Toby Murray</td>
<td>Verified Compilation of Noninterference for Shared-Memory Concurrent Programs</td>
</tr>
<tr>
<td>15:00-15:30</td>
<td>Stefan Brunthaler</td>
<td>Software Diversity vs. Side Channels</td>
</tr>
<tr>
<td>15:30-16:00</td>
<td>Kedar Namjoshi</td>
<td>Plugging Leaks Introduced by Compiler Optimizations</td>
</tr>
<tr>
<td>16:00-16:30</td>
<td></td>
<td>cake</td>
</tr>
<tr>
<td>16:30-18:00</td>
<td>Lead: Amal Ahmed</td>
<td>Discussion: Formal verification and proof techniques</td>
</tr>
</tbody>
</table>

**Friday**

<table>
<thead>
<tr>
<th>Time</th>
<th>Speaker</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>07:30-09:00</td>
<td></td>
<td>breakfast</td>
</tr>
<tr>
<td>09:00-09:30</td>
<td>Stephanie Weirich</td>
<td>Verifying the Glasgow Haskell Compiler Core language</td>
</tr>
<tr>
<td>09:30-09:50</td>
<td>Gabriele Keller</td>
<td>Data Refinement for Cogent</td>
</tr>
<tr>
<td>09:50-10:20</td>
<td>Frédéric Besson</td>
<td>Preservation of safe erasure as an information flow property</td>
</tr>
<tr>
<td>10:20-10:50</td>
<td></td>
<td>coffee</td>
</tr>
<tr>
<td>10:50-11:10</td>
<td>Tamara Rezk</td>
<td>A project on secure compilation in the context of the IoT</td>
</tr>
<tr>
<td>11:10-11:30</td>
<td>Cédric Fournet</td>
<td>Building Secure SGX Enclaves using F*, C/C++ and X64</td>
</tr>
<tr>
<td>11:30-11:50</td>
<td>Vincent Laporte</td>
<td>Secure compilation of side-channel countermeasures: the case of cryptographic “constant-time”</td>
</tr>
<tr>
<td>12:15-14:00</td>
<td></td>
<td>lunch</td>
</tr>
</tbody>
</table>
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 results to date. The talk will give some brief background to the ideas, show how they play out in practice, and open a broader discussion as to how this insight could inform compiler correctness research.

In the CHERI JHJ work presented at APLAS last year, we demonstrated one possible way of embedding in F*.

Most compiler programs contain a mixture of different languages, 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 implementation.

Where we are and where we go:
- Low-level code
- LLVM
- Recent work on “miscompilation” uses hardware-related techniques to monitor individual machine operations. This talk will sketch preliminary ideas for how to raise the definition of low-level code to the level of C code. C-level policies should be used to express high-level policies that are false or impossible to specify at machine level (e.g., information flow control or compartmentalization) and to enforce particular variants of C-language policies (e.g., dealing with faults of memory safety based on different pointer aliasing rules). C-level policies can be (verifiably) compiled into machine policies to be enforced by existing (pointee) monitors.

Preserving high-level guarantees in the presence of low-level code
- The talk will give some brief background to the ideas, show how they play out in example, and open a broader discussion as to how this insight could inform compiler correctness research.

Establishing that two programs are contextually equivalent is hard, yet essential for reasoning about semantics-preserving program transformations such as optimizations and code specialization. The Velm project aims to use Coq to formalize and reason about LLVM program transformations and as part of this work we are using a variant of Lazy call-by-push-value language. I will talk about how we establish the soundness of an equational theory for call-by-push-value and how we used our equational theory to significantly simplify the verification of classic optimizations.

Theorem 1: There is no low-level code
- 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 can also be done by a source-level attacker. This means that programmers need to take complete control over the component and use its privileges to attack the remaining uncompromised components.

In this paper we study an aspect of IR design that has received little attention: the role of components in modular verification of multi-pass compilers.

In CHERI, one possible way is to enforce either the LLVM or X64 semantics (e.g. differing flavors of memory safety based on differing pointer aliasing rules). C-language policies can be (verifiably) compiled into machine policies to be enforced by existing (pointee) monitors.

I will end with an insight for those working on compositional results that require “weaker” precision of compiled components than it is provided by a low-level language. In such cases, the high-level policies can be (verifiably) compiled to sound low-level policies to be enforced by an existing (pointee) monitor.

In the presence of low-level code, there is no low-level code.
David Chisnell
Teaching a production compiler that integers are not pointers
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/++ abstract machine that maintain compatibility with large corpora of real-world code while preserving memory safety.

Khilen Gubbi, Alex Richardson, Peter Sewell
15 + 15

Deepak Garg
What is secure compilation?
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 loop 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 FacCT, a new domain-specific language that addresses the challenge of writing constant-time crypto code. While FacCT developers write crypto code using standard high-level language constructs, FacCT, in turn, compiles such high-level code into constant-time C code. In particular, I will discuss how we have integrated FacCT in several existing, large projects and languages (e.g., Python, and Haskell).

Catrin Hinch (attending), Pascal Paolini (also attending), Cermin Adde, Jeremy Tissot
40 + 20 tutorial (very few data)

Deian Stefan
Constant-time crypto programming with FacCT
In the Rosetta project, we have been building frontends for understanding the safety claims of the Rosetta language and for evolving the language safety. In so doing, we have thus far assumed a memory model in which the only form of undefined behavior is data races and memory safety violations. However, this is too simplistic. The Rosetta developers would like to support more aggressive compiler optimizations that exploit non-executing 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-executing assumptions. In this talk, I will report on several avenues currently being explored for defining undefined behavior in Rust.

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.

Secure computing requires enforcing secrets to limit the prudicity for an attacker to probe the content of memory. At source level, secrets are typically performed by a method (secreted). Yet, as secret is dead, compiler optimisations may remove this piece of code and therefore break the security.

Thomas Van Strydonck (not attending), Frank Piessens, Lau Losholt Engberg (not attending), Lars Birnild, Abram Ekroth, Stelios Tsampas (not attending), Marco Paolini, Deepak Garg
20 + 20

Frederic Bienzon
Preservation of safe use as an information flow property
In the talk, I will test the audience a semantic definition of preservation of safe use 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.

Ralf Jung
15 + 15

Frederic Bienzon
Capability machines as a target for secure compilation
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. Furthermore, 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.

Christine Rizkallah
10+10

Gabrielle Koller
Data Refinement for Coparent
This talk will present Secure Virtual Architecture (SVA), a virtual instruction set computing architecture which we have used to enforce security policies on both application and operating system kernel code. We 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 our approach to write secure compilation techniques into SVA to further reduce its already small trusted computing base size, and to discuss the use of secure compilation techniques in operating system code.

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 secure by reasoning about the transformation as a predicate transformation, and expressing a refinement relation (a "forbid") that is produced exactly.

Christine Rizkallah
10+10

John Ciesiell
Virtual Instruction Set Computing with Secure Virtual Architecture
This talk will present Secure Virtual Architecture (SVA), a virtual instruction set computing architecture which we have used to enforce security policies on both application and operating system kernel code. We 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 our approach to write secure compilation techniques into SVA to further reduce its already small trusted computing base size, and to discuss the use of secure compilation techniques in operating system code.

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 secure by reasoning about the transformation as a predicate transformation, and expressing a refinement relation (a "forbid") that is produced exactly.

Christopher Rizkallah
10+10

Kedar Namjoshi
Plugging Leaks Introduced by Compiler Optimizations
There are several open question (e.g., how to explain 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
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, which supports features such as untagged unions and untagged memory, all of the way through the compilation pipeline to hardware that can preserve this distinction at run-time.

Scott Owens (attending), Ramana Kumar, Michael Noram, Yong Kong Tan, Anthony Fox
15 + 15

Magnus Myreen
Is the verified CakeML compiler secure?
Energy harvesting enables novel devices and applications without batteries. However, this is too simplistic. The Rust developers would like to assume a memory model in which the only form of undefined behavior is 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-executing 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-executing assumptions. In this talk, I will report on several avenues currently being explored for defining undefined behavior in Rust.

Thomas Van Strydonck (not attending), Frank Piessens, Lau Losholt Engberg (not attending), Lars Birnild, Abram Ekroth, Stelios Tsampas (not attending), Marco Paolini, Deepak Garg
20 + 20

Norrish, Yong Kiam Tan, Anthony Fox

Piessens, Lau Skorstengaard (not attending), Lars Birnild, Abram EKroth, Stelios Tsampas (not attending), Marco Paolini, Deepak Garg
20 + 20

Thomas Van Strydonck (not attending), Frank Piessens, Lau Losholt Engberg (not attending), Lars Birnild, Abram Ekroth, Stelios Tsampas (not attending), Marco Paolini, Deepak Garg
20 + 20

Ralf Jung
15 + 15

Ralf Jung
15 + 15

Christine Rizkallah
10+10

Christopher Rizkallah
10+10

Norrish, Yong Kiam Tan, Anthony Fox

Piessens, Lau Skorstengaard (not attending), Lars Birnild, Abram EKroth, Stelios Tsampas (not attending), Marco Paolini, Deepak Garg
20 + 20

Thomas Van Strydonck (not attending), Frank Piessens, Lau Losholt Engberg (not attending), Lars Birnild, Abram Ekroth, Stelios Tsampas (not attending), Marco Paolini, Deepak Garg
20 + 20

Ralf Jung
15 + 15

10+10

10+10

15 + 15

15 + 15

15 + 15

15 + 15

10+10

15 + 15
Abstract (can be informal)

Collaborators (especially if also attending)

Zoe Paraskevopoulou
Xavier Leroy
Toby Murray
Tamara Rezk
Stephanie Weirich
Santosh Nagarakatte
Peter Sewell
Pranab Bhattacharyya
Santosh Nagarakatte
Stefan Bürhler
Stephanie Weirich
Steve Zdancewic
Tamar Reznik
Toby Murray
Xavier Lamy
Zoe Paraskevopoulou

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 interpolation 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 compile code from low-level attackers can be given two dual specifications with respect to the refinement relation. An interpolation function is sound if its output refines its input, and "optimal" if it has the least behavior of any refinement of the input. Dually, a protection function is sound if it 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 "inective", which is here equivalent to saying that "enforce is protect" if the refinement relation is sound.

This talk presents a simple case of a "galois connection" based approaches to security, but we argue that 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

Specifications for Dynamic Enforcement of Relational Program Properties

Thoughts on preserving abstractions

Short update for several related projects under our REMS umbrella focusing on the bits most relevant to secure compilation:

a) Analyses and optimizations at the level of abstract syntax trees, a.k.a. SSA, and also related work on the Glasgow Haskell Compiler Core language
b) Formalizing and understanding the endpoints of compiler transformations via equational theory based on the usual notions of beta-equivalence. By relating 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 optimizations can erroneously remove the instrumentation

c) proving security properties of Cheri compiler

Peter Sewell
10 + 10

Secure Compilation – understanding the endpoints?

Is there a Semantic Mismatch?

Compiler Optimizations with Retrofitting/Transformation: How can we transport reasoning and program transformations from one level to another?

In this talk, I will first present our work on SGBounds on how to achieve lightweight memory safety in the context of SGX Enclaves.


Pranab Bhattacharyya
15 + 15

Memory safety for Shielded Execution

Stefan Bürhler
15 + 15

Software Diversity vs. Side Channels

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 optimizations pass 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 troco-zoo tool to transform the source code of GHC into Haskell. 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 optimizations can erroneously remove the instrumentation

Stephanie Weirich
15 + 15

Verifying the Glasgow Haskell Compiler Core language

Call-by-Push-Value and Reasoning about Low-Level IRs

This talk will explain our on-going work on the LLVM IR.

Christine Rizkallah (attending – she will talk about a different, but related piece of this project)

Christine Rizkallah
15 + 15

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 opportunities on how to address them.

Tamar Reznik
10+10

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 some of the initial implementation in the context of a compiler from a simple White language to an idealized 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.

Toby Murray
30 + 20

Why compiler correctness says so little about security properties

Compiler transformations may fail to preserve the resource consumption of compiled programs. A notable example is 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 to reason about low-level logical relations that allow 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

Zoe Paraskevopoulou

Closure Conversion is Safe-for-Space