Report from Dagstuhl Seminar 18201

Secure Compilation

Edited by

Amal Ahmed[1], Deepak Garg[2], Catalin Hritcu[3], and Frank Piessens[4]

1 **Northeastern University and INRIA – Paris, FR,** amal@ccs.neu.edu
2 **MPI-SWS – Saarbrücken, DE,** dg@mpi-sws.org
3 **INRIA – Paris, FR,** catalin.hritcu@gmail.com
4 **KU Leuven, BE,** frank.piessens@cs.kuleuven.be

Abstract

Secure compilation is an emerging field that puts together advances in security, programming languages, verification, systems, and hardware architectures in order to devise secure compilation chains that eliminate many of today's vulnerabilities. Secure compilation aims to protect a source language's abstractions in compiled code, even against low-level attacks. For a concrete example, all modern languages provide a notion of structured control flow and an invoked procedure is expected to return to the right place. However, today's compilation chains (compilers, linkers, loaders, runtime systems, hardware) cannot efficiently enforce this abstraction: linked low-level code can call and return to arbitrary instructions or smash the stack, blatantly violating the high-level abstraction. The emerging secure compilation community aims to address such problems by devising formal security criteria, efficient enforcement mechanisms, and effective proof techniques.

This seminar strived to take a broad and inclusive view of secure compilation and to provide a forum for discussion on the topic. The goal was to identify interesting research directions and open challenges by bringing together people working on building secure compilation chains, on developing proof techniques and verification tools, and on designing security mechanisms.

Seminar May 13–18, 2018 – http://www.dagstuhl.de/18201
2012 ACM Subject Classification Security and privacy → Formal security models
Keywords and phrases secure compilation, low-level attacks, source-level reasoning, attacker models, full abstraction, hyperproperties, enforcement mechanisms, compartmentalization, security architectures, side-channels
Digital Object Identifier 10.4230/DagRep.8.5.1
Edited in cooperation with Roberto Blanco (INRIA – Paris, FR)

1 Executive Summary

Amal Ahmed (Northeastern University and INRIA – Paris, FR)
Deepak Garg (MPI-SWS – Saarbrücken, DE)
Catalin Hritcu (INRIA – Paris, FR)
Frank Piessens (KU Leuven, BE)

Today's computer systems are distressingly insecure. The semantics of mainstream low-level languages like C and C++ is inherently insecure, and even for safer languages, establishing security with respect to a high-level semantics does not prevent devastating low-level attacks. In particular, all the abstraction and security guarantees of the source language are currently lost when interacting with lower-level code, for instance when using low-level libraries. For a

Secure Compilation, *Dagstuhl Reports*, Vol. 8, Issue 05, pp. 1–30
Editors: Amal Ahmed, Deepak Garg, Catalin Hritcu, and Frank Piessens
Dagstuhl Reports
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

concrete example, all modern languages provide a notion of structured control flow and an invoked procedure is expected to return to the right place. However, today's compilation chains (compilers, linkers, loaders, runtime systems, hardware) cannot efficiently enforce this abstraction: linked low-level code can call and return to arbitrary instructions or smash the stack, blatantly violating the high-level abstraction.

Secure compilation is an emerging field that puts together advances in security, programming languages, compilers, verification, systems, and hardware architectures in order to devise secure compiler chains that eliminate many of today's low-level vulnerabilities. Secure compilation aims to protect high-level language abstractions in compiled code, even against low-level attacks, and to allow sound reasoning about security in the source language. The emerging secure compilation community aims to achieve this by:

1. **Identifying and formalizing secure compilation criteria and attacker models.** What are the properties we want secure compilers to have, and under what attacker models? Should a secure compilation chain preserve observational equivalence of programs? Should it preserve some class of security properties of the source programs? Should it guarantee invariants on the run-time state of the compiled program (like for instance well-formedness of the call-stack)? And what are realistic attacker models? Can attackers only interact with compiled programs by providing input and reading output? Or can they link arbitrary low-level code to the program? Well-studied notions like fully abstract compilation provide partial answers: a fully abstract compiler chain preserves observational equivalence under an attacker model where attackers are target-level contexts. Even where this is the desired end-to-end security goal, it can still be too hard to enforce, for instance in cases where target level contexts can measure time.
2. **Efficient enforcement mechanisms.** The main reason today's compiler chains are not secure is that enforcing abstractions in low-level compiled code can be very inefficient. In order to overcome this problem, the secure compilation community is investigating various efficient security enforcement mechanisms: from the use of static checking of low-level code to rule out linking with ill-behaved contexts, to software rewriting (e.g., software fault isolation), dynamic monitoring, and randomization. One key enabler is that hardware support for security is steadily increasing.
3. **Developing effective formal verification techniques.** Secure compilation properties like full abstraction are generally much harder to prove than compiler correctness. Intuitively, in order to show full abstraction one has to be able to back-translate any low-level context attacking the compiled code to an equivalent high-level context that can attack the original source code. This back-translation is, however, nontrivial, and while several proof techniques have been proposed (e.g., based on logical relations, bisimulations, game semantics, multi-language semantics, embedded interpreters, etc.), scaling these techniques to realistic secure compilers is a challenging research problem. This challenge becomes even more pronounced if one expects a strong level of assurance, as provided by formal verification using a proof assistant.

The Secure Compilation Dagstuhl Seminar 18201 attracted a large number of excellent researchers with diverse backgrounds. The 45 participants represented the programming languages, formal verification, security, and systems communities, which led to many interesting points of view and enriching discussions. Some of these discussions were ignited by the "guided discussions" on the 3 aspects above and by the 35 talks contributed by the participants. The contributed talks spanned a very large number of topics: investigating

various secure compilation criteria and attacker models, building prototype secure compilation chains, proposing different enforcement techniques, studying the relation to verified compilation and compositional compiler correctness, specifying and restricting undefined behavior, protecting against side-channels, studying intermediate representations, performing translation validation, securing multi-language interoperability, controlling information-flow, compartmentalizing software, enforcing memory safety, compiling constant-time cryptography, securing compiler optimizations, designing more secure (domain-specific) languages, enforcing security policies, formally specifying the semantics of realistic languages and ISAs, compartmentalization, capability machines, tagged architectures, integrating with existing compilation chains like LLVM, making exploits more difficult by diversification, multi-language interoperability, etc. Talks were interspersed with lively discussions, since by default each speaker could only use half of the time for presenting and had to use the other half for answering questions and engaging with the audience.

Given the high interest spurred by this first edition and the positive feedback received afterwards, we believe that this Dagstuhl Seminar should be repeated in the future. Particular aspects that could still be improved in future editions is focusing more on secure compilation and spurring more participation from the practical security and systems communities.

2 Table of Contents

3 Guided Discussions

3.1 What Is Secure Compilation? Security Goals and Attacker Models

Discussion led by Catalin Hritcu (INRIA – Paris, FR)

Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.CatalinHritcu.Slides.pdf

In the broadest sense, the goal of secure compilation research is to devise more secure compilation chains. Since there are many different ways to define "more secure," there are also many different notions of secure compilation. This discussion was aimed at identifying various different security goals and attacker models for secure compilation chains. Here we use the term "compilation chain" to include not just the compiler, but also the linker, loader, runtime, operating system, hardware, and security enforcement mechanisms at any of these levels. We do this since the responsibility of enforcing secure compilation often does not rest just with the compiler, but is shared by various parts of the compilation chain. For instance, achieving memory safety requires not only changing the compiler, but also most other components of the compilation chain have to at least be taught that pointers are not integers, and to achieve efficient enforcement the hardware needs to be extended as well.

So what are some of the possible security goals and attacker models for secure compilation chains? A first class of secure compilation chains aim at providing a "safer" semantics for unsafe low-level languages like C and C++, whose standard semantics call out a large set of undefined behaviors for which compilers can produce code that behaves arbitrarily, often leading to exploitable vulnerabilities. For instance, memory safety is aimed at turning spatial and/or temporal memory violations–e.g., buffer overflows, use after free–into safe behavior–e.g., raising an exception or terminating the program. Similarly, type safety can ensure that invalid casts are always detected and do not cause undefined behavior. The standard attacker model for type and memory safety protects against an external adversary that provides malicious, often malformed, inputs into the program and tries to hijack control, corrupt or disclose data, etc.

Ideally, one would like to turn as much undefined behavior in C and C++ as possible into safe behavior. However, especially when done solely in software, this can have a very high performance cost. So most security defenses that are widely deployed today are mitigations focused not at making languages like C and C++ safe, but instead at making exploiting security vulnerabilities more difficult: control-flow integrity, data-flow integrity, code-pointer integrity, lightweight stack protection, randomization. The attacker model for these mitigations is that the attacker can send inputs that exploit a particular class of vulnerabilities: for instance the attacker can use a buffer overflow to access memory via say contiguous writes or arbitrary reads. The goal of the attacker is then to inject code or behavior, to corrupt or leak data, while avoiding the mitigations in place. With enough effort a motivated attacker can usually achieve just this, and the goal here is only to increase the attacker effort, not to provide watertight guarantees.

In contrast, compartmentalization (e.g., software-fault isolation) is a mitigation technique that does provide watertight guarantees. The security goal of compartmentalization is to limit the damage of an attack only to the compromise of the components encountering undefined behavior. In particular, compartmentalization can be applied in unsafe low-level languages to structure large, performance-critical applications into mutually distrustful components that have clearly specified privileges and interact via well-defined interfaces. Intuitively,

protecting each component from all the others should bring strong security benefits, since a vulnerability in one component need not compromise the security of the whole application.

The applications of compartmentalization are, however, much broader. One can use compartmentalization to, for instance: (1) protect a trusted host application from untrusted plugins or libraries that could be malicious (e.g., as usually done for securing securing web browsers plugins, etc.); (2) protect a secure enclave from a malicious host (e.g., Intel SGX, ARM TrustZone, Sancus, Sanctum, etc.); or (3) protect mutually distrustful components written in an unsafe language against each other (e.g., as done for achieving least privilege design with process-based isolation, SFI, capability machines, tagged architectures, etc). In all these scenarios a minimal security goal is to preserve the integrity of the code and data of each component from malicious or compromised code in the other components. In addition one could also aim that no component can infer the secrets of other components, other than communicating with them through their high-level interface. This is particularly challenging though when bad components can also observe side-channels like execution time. Finally, one could also aim at protecting the availability of critical components, ensuring that others cannot cause crashes or hangs.

The common way of formalizing the security guarantees of compartmentalization is in terms of the source-level security reasoning principles it enables. Reasoning about the security in the source language (or "the safe part" of the source language, without undefined behaviors) is useful because then one does not need to worry about low-level attacks that can only happen at the target level, since one knows the abstractions of the source language are implemented in a watertight way by the secure compilation chain. A good way to formalize this is in terms of preserving various classes of security property during compilation: (1) trace properties such as safety and liveness, (2) hyperproperties such as noninterference, or (3) relational hyperproperties such as trace equivalence and observational equivalence.

An important point in the discussion was that specially designed source languages or source language extensions could make it easier to precisely specify the intended security properties, so that the secure compilation chain only needs to preserve those, and can thus be more efficient than if trying to preserve a large class of properties. For instance, explicitly annotating what is the secret data that external observers or other components should not be able to obtain, maybe even using side-channels like timing, gives the compilation chain the freedom to more efficiently handle any data that is not influenced by secrets.

We end this report summary with some interesting questions raised in our discussion:

- What are meaningful security properties to preserve in a particular application domain?
- When is it more meaningful for secure compilation chains to preserve large classes of properties (in which case, one doesn't don't need to specify much at the source level), and when is it more meaningful to preserve application-specific security properties?
- How much can program verification help and what are its scalability limitations?
- How does one go about preserving security intent all the way to the hardware and how can one convince hardware manufacturers to use this information for security?
- Can domain-specific languages make certain properties easier to achieve?
- In cases where security is not just binary, can we properly quantify the notion of attacker cost, taking maybe inspiration from cryptographic proofs?
- What are low-cost mitigations for (the lack of) full abstraction, maybe inspired by current mitigations for (the lack of) memory safety?

3.2 Effective Enforcement Mechanisms for Secure Compilation

Discussion led by Frank Piessens (KU Leuven, BE)

Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.FrankPiessens.Slides.pdf

This discussion session started from the observation that there is a wide variety of enforcement mechanisms, including hardware based mechanisms (such as processor privilege levels, virtual memory, capabilities, trusted computing, ...), software based mechanisms (including techniques such as type checking, static analysis, program verification, run-time monitoring, taint tracking, ...), and cryptographic mechanisms.

A first question that was discussed is where this wide variety of techniques is still insufficient. Several areas where there is need for novel kinds of enforcement mechanisms were discussed, the most prominent being the area of protecting against micro-architectural side-channel attacks.

A second topic that was addressed during the discussions is the issue of passing security information across abstraction layers, and in particular the question of what security information should be passed down to the compiler and further down to the hardware by software source code. Should software engineering inspired abstraction mechanisms be enforced as security boundaries after compilation? Or should specific security annotations be added to the source code to inform the compiler and the hardware about what security boundaries to enforce?

Finally, the discussion focused on trade-offs (for instance, expressivity versus performance versus complexity) of different enforcement techniques.

3.3 Formal Verification and Proof Techniques for Secure Compilation

Discussion led by Amal Ahmed (Northeastern University and INRIA – Paris, FR)

Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.AmalAhmed1.Slides.pdf

This session involved discussion of what kinds of formalisms and proof techniques might be needed for verifying secure compilation and compositional compiler correctness.

The first issue discussed was what it would take to extend existing verified compilers into secure compilers and compositionally correct compilers. Here "secure compiler" might encompass various different notions of security, e.g., resistant to side-channel attacks, satisfying robust safety preservation, fully abstract, and so on. Two central questions raised were: (1) when do we need entirely new proof architectures, and (2) what are good strategies for reusing mechanized proof efforts. The following points were raised during this part of the discussion:

- Taking CompCert as an exemplar, it was posited that a central challenge is stating the security properties we want. A related issue (discussed earlier at the seminar) is that C is not a language in which programmers can express their "security intent" so we either need to (a) have the compiler writer decide what security properties to enforce or (b) provide programmers with compiler flags or program annotations so they can communicate their security intent to the compiler.

- Proof architectures such as CompCert's have proved fairly reusable, but whether we can keep using refinement and simulation style proofs will likely depend on the security properties we wish to establish.
- There's a question of what security architectures are employed for enforcing security properties in a compiler like CompCert, e.g., capability machines like CHERI or tag-based architectures. There may be potential difficulties here since these mechanisms may be quite different from CompCert's memory model.
- A significant issue that must be taken into account when extending existing verified compilers into secure compilers is that the correctness of certain compiler optimizations will be influenced by the attacker model. This might complicate the statements of theorems as well as the proofs themselves.

The second issue discussed was about better techniques for compositional compiler correctness. While there are a number of existing techniques–e.g., multi-language semantics, cross-language logical relations, PILS, interaction semantics with structured simulations–it would be useful to have guidelines about which technique is suitable when. One example is that while the language-independent interaction semantics used by Compositional CompCert works for CompCert–where the source, intermediate, and target languages use the same memory model–it's unclear how to extend it to compilers where the languages have different memory models. There are also open questions of how to reduce the effort involved in multi-pass compilers–i.e., making it easier to prove transitivity (or vertical compositionality) for a compositionally correct compiler–and the related question of how to do prove transitivity when different passes of the compiler are verified using different proof techniques. Finally, it would be nice to have some common infrastructure for mechanizing proofs. The following additional points were raised in the discussion:

- Compositional compiler correctness requires reasoning about the behavior of the target-level (assembly) code that a compiled component is linked with. But if we take that target code to simply be assembly then we may have a difficult reasoning problem, as well as a highly powerful attacker model.
- Can we impose constraints on the attacker even when it is assembly code? Yes, we can either impose constraints statically or dynamically through mechanisms such as SFI, capabilities, or putting code in enclaves as in SGX.
- Once target contexts (attackers) are somewhat constrained using static or dynamic mechanisms, we must correctly model the power of target-level attackers. The multi-language semantics approach gives one way to do this. Different kinds of multi-language semantics can be set up to allow more restricted or less restricted interactions between target contexts and source (or compiled) components. Another strategy might be to come up with the right abstractions to add to the source level that correctly model the additional power of the target contexts/attackers. This is similar to the "linking types" idea presented at the seminar.
- The ultimate goal for secure compilation is to have compiled code that does not have attacks. If we could lift that specific security goal into a proof obligation at the source level, what would that proof obligation be? If we can specify what abstractions we must add to the source to model additional attacker power, then at least we know exactly what we must protect our source components against.
- Over time, as we encounter new attacks, we may want to adapt our proofs of compositional compiler correctness or secure compilation to new attacker models. But this will probably be quite challenging since this is an instance where the tension between horizontal and vertical compositionality might come into play.

The final issue brought up was regarding proof methods that help reason about the power of the adversary in secure compilation. In particular, context-based back-translation has been widely used, but back-translation techniques differ depending on how different the source and target languages are from each other, whether they are Turing-complete or not, and whether the back-translation used is syntax-based or trace-based. Trace-based back-translations might be easier to reuse across languages.

4 Working Groups

4.1 Meltdown and Spectre Attacks

Discussion led by Chris Hawblitzel (Microsoft Research – Redmond, US-WA)

Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.ChrisHawblitzel.Slides.pptx

The discussion was illustrated by a (contrived) Spectre example. Consider an array of flat pointers, not secret, each pointing to an integer, also not secret. In memory, the array is followed by some secret data. In performing a standard, safe loop over the elements of the array, the hardware may speculatively go beyond the bounds of the array and execute the next potential iteration, therefore loading the secret into the cache and exposing it to an adversary.

For the described example, potential mitigations include the modular indexing of elements in the array, which are then accessed modulo the length of the array. Attempts to implement such fixes can be performed at the source level or directly in the compiler (or even on hardware)–it should be noted that interval analysis, already implemented, say, in JavaScript compilers, allows a compiler to patch up this vulnerability.

The discussion revolved around two main questions:

4.1.1 What Can Software Do?

Several ideas were given:

- For conditional branches: clamping or sandboxing array accesses (as in the motivating example).
- For unconditional jumps: turning speculation off, other measures?
- In general, formal reasoning requires a hardware model: e.g., an operational semantics that nondeterministically speculates on conditional branches (however, the necessity of establishing a fruitful dialogue with architects was noted). A suggested example would involve adding a cache to the operational semantics and stating properties about that cache. The obvious question will be whether a model is good enough to prevent attacks in practice.

Regardless, it was noted that speculation may not need to be turned off completely, though these considerations may be application-dependent.

4.1.2 What Can Hardware Do?

Again, several lines of discussion were considered:

- Partitioning of shared resources: the software would decide which data goes into which partition (and the hardware would be responsible for providing partitions, each with its own cache and resources).
- Protection of secret data: the software would mark some addresses and data as sensitive.
- Side channel avoidance without software help: a more speculative idea, by which the hardware would avoid committing changes to shared resources until all relevant speculation were resolved.

As in the discussion of software-based strategies, the need to engage in discussion with more architects was noted.

An upcoming *Panel on the implications of the Meltdown & Spectre design flaws* (http://iscaconf.org/isca2018/panel.html) was mentioned, where the state of affairs was tentatively summarized as:

> Computer Architecture 1.0 specifies the timing-independent functional behavior of a computer, while Micro-Architecture is the implementation techniques that improve performace. What if a computer that is completely correct by Architecture 1.0 can be made to leak protected information via timing, a.k.a., Micro-Architecture?

4.2 C Semantics in Depth

Discussion led by Peter Sewell (University of Cambridge, GB)

The working group revolved around a discussion of pointer provenance and uninitialised reads in the C programming language. The attendance included the following participants:

- Frédéric Besson
- David Chisnall
- John T. Criswell
- Chung-Kil Hur
- Xavier Leroy
- Santosh Nagarakatte
- Steve Zdancewic
- Roberto Blanco
- Daniel Patterson
- Andrew Tolmach
- Peter Sewell

5 Overview of Talks

5.1 Compositional Compiler Correctness and Secure Compilation: Where We Are and Where We Want to Be

Amal Ahmed (Northeastern University – Boston, US and INRIA - Paris, FR)

Joint work of Amal Ahmed, Daniel Patterson
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.AmalAhmed.Slides.pdf

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 abstract compilation: when it comes to proving such compilers correct, truly modular verification of multi-pass compilers seems impossible.

5.2 Thoughts on Preserving Abstractions

Nick Benton (Facebook Research – London, GB)

The talk discussed the principle that high-level reasoning as performed by a programmer or a compiler should remain valid when the compiled code runs in a real environment. To this end, the behavioral characterization of interface contracts should be independent of the source language and the compiler, modulo calling and linking conventions. It is thus necessary to agree on a language to express the aforementioned interface contracts. The end result of a high-level denotational semantics with a low-level operational behavior will be a proof obligation to show that a piece of code behaves like some given mathematical function.

When reasoning about preservation of abstractions, it was observed that there is always, in fact, an appeal to some form of denotational semantics, whether this is explicitly acknowledged or not. In current practice, many such treatments are not fully abstract, but nonetheless, weaker, "good enough" notions of abstraction are routinely used to good effect, provided that they offer sufficient abstraction for the task at hand–in fact, some of the finer points of full abstraction are often not very useful in practice, nor are they well-understood in their full generality.

Finally, a mixed-language approach to the problem was discussed, noting the risk of breaking abstractions too severely, and the difficulties and costs incurred by some potential mitigations to that risk. However, the difficulties of preserving abstractions express themselves fully in the presence of higher-order and/or fairly strong notions of purity, whereas most foreign function interfaces are, rather, first-order in nature–and when they are not, the type system offers assistance in those parts that cross the boundary.

5.3 Secure Compilation of Safe Erasure

Frédéric Besson (INRIA – Rennes, FR)

Joint work of Frédéric Besson, Thomas Jensen, Alexandre Dang
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.Fr%C3%A9d%C3%A9ricBesson.Slides.pdf

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 tested on the audience a semantics definition of (preservation) of safe erasure phrased in terms of quantitative information flow. I then sketched how typical compiler optimisations (DSE, register allocation) need to be modified to preserve this property.

5.4 CompCertSFI

Frédéric Besson (INRIA – Rennes, FR)

Joint work of Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas Jensen

We describe the design, implementation and proof of an efficient, machine-checked CompCert implementation of Portable Software Fault Isolation. We propose a novel sandboxing transformation that has a well-defined C semantics and which supports arbitrary function pointers. Our experiments show that our formally verified technique is a competitive way of implementing Software Fault Isolation.

5.5 Memory Safety for Shielded Execution

Pramod Bhatotia (The University of Edinburgh, GB)

Joint work of Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, Christof Fetzer
Main reference Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, Christof Fetzer: "SGXBOUNDS: Memory Safety for Shielded Execution", in Proc. of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017, pp. 205–221, ACM, 2017.
URL http://dx.doi.org/10.1145/3064176.3064192

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

I will conclude the talk with our on-going work on Intel MPX Explained: https://intel-mpx.github.io/

5.6 Software Diversity vs. Side Channels

Stefan Brunthaler (Universität der Bundeswehr – Munich, DE)

Main reference Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, Michael Franz: "Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity", in Proc. of the 22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015, The Internet Society, 2015.

URL https://www.ndss-symposium.org/ndss2015/thwarting-cache-side-channel-attacks-through-dynamic-software-diversity

Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.StefanBrunthaler.Slides.pdf

The past couple of years have seen attacks becoming increasingly sophisticated, primarily due to the discovery and incorporation of side channels. Among others, 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, I will present new ideas and results of either mitigating or substantially lessening the impact of these side-channel attacks.

5.7 Preserving High-Level Invariants in the Presence of Low-Level Code

David Chisnall (University of Cambridge, GB)

Joint work of David Chisnall, 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, Robert N. M. Watson

Main reference David Chisnall, Brooks Davis, Khilan Gudka, David Brazdil, Alexandre Joannou, Jonathan Woodruff, A. Theodore Markettos, J. Edward Maste, Robert Norton, Stacey D. Son, Michael Roe, Simon W. Moore, Peter G. Neumann, Ben Laurie, Robert N. M. Watson: "CHERI JNI: Sinking the Java Security Model into the C", in Proc. of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017, pp. 569–583, ACM, 2017.

URL http://dx.doi.org/10.1145/3037697.3037725

Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.DavidChisnall.Slides.pptx

Most complex 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 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.

5.8 Teaching a Production Compiler That Integers Are Not Pointers

David Chisnall (University of Cambridge, GB)

Joint work of David Chisnall, 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, Robert N. M. Watson
Main reference David Chisnall: "No such thing as a general-purpose processor", Commun. ACM, Vol. 57(12), pp. 44–48, 2014.
URL http://dx.doi.org/10.1145/2677030
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.DavidChisnall1.Slides.pptx

Over the past six years, we have 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.

5.9 Virtual Instruction Set Computing with Secure Virtual Architecture

John Criswell (University of Rochester, GB)

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 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.

5.10 Capability Machines as a Target for Secure Compilation

Dominique Devriese (KU Leuven, BE)

Joint work of Dominique Devriese, Thomas Van Strydonck, Frank Piessens, Lau Skorstengaard, Lars Birkedal, Akram El-Korashy, Stelios Tsampas, Marco Patrignani, Deepak Garg
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.DominiqueDevriese.Slides.pdf

A quick introduction to capability machines, and an overview of ideas about how different properties can be enforced using different extensions of capability machines

5.11 Defining Undefined Behavior in Rust

Derek Dreyer (MPI-SWS – Saarbrücken, DE)

Joint work of Derek Dreyer, Ralf Jung
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.DerekDreyer.Slides.pdf

In the RustBelt project, we have been building foundations for understanding the safety claims of the Rust programming 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.

5.12 Compiling a Cecure Variant of C to Capabilities

Akram El-Korashy (MPI-SWS – Saarbrücken, DE)

Joint work of Akram El-Korashy, Dominique Devriese, Deepak Garg, Marco Patrignani, Frank Piessens, Stelios Tsampas
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.AkramEl-Korashy.Slides.pdf

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.

5.13 Building Secure SGX Enclaves using F*, C/C++ and X64

Cédric Fournet (Microsoft Research – Cambridge, GB)

Joint work of Cédric Fournet, Anitha Gollamudi

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*.

5.14 How to Define Secure Compilation? (A Property-Centric View)

Deepak Garg (MPI-SWS – Saarbrücken, DE)

Joint work of Deepak Garg, Carmine Abate, Roberto Blanco, Catalin Hritcu, Marco Patrignani, Jérémy Thibault
Main reference Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, Jérémy Thibault: "Exploring Robust Property Preservation for Secure Compilation", CoRR, Vol. abs/1807.04603, 2018.
URL http://arxiv.org/abs/1807.04603
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.DeepakGarg.Slides.pdf

This talk presents a possible approach to defining compiler security as the preservation of security properties despite adversarial contexts. The talk starts from the idea that compiler correctness can be defined as preservation of properties (in the absence of adversaries). Adversarial contexts are then introduced, and a notion of compiler security, parametrized by a class of security properties, is defined. Particularly interesting classes include safety properties, hyperproperties (e.g., non-interference), and relational hyperproperties (e.g., observational equivalence).

5.15 When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise

Catalin Hritcu (INRIA – Paris, FR)

Joint work of Catalin Hritcu, Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Théo Laurent, Benjamin C. Pierce, Marco Stronati, Andrew Tolmach
Main reference Guglielmo Fachini, Catalin Hritcu, Marco Stronati, Arthur Azevedo de Amorim, Ana Nora Evans, Carmine Abate, Roberto Blanco, Théo Laurent, Benjamin C. Pierce, Andrew Tolmach: "When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise", CoRR, Vol. abs/1802.00588, 2018.
URL http://arxiv.org/abs/1802.00588
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.CatalinHritcu1.Slides.pdf

We propose a new formal criterion for evaluating secure compilation schemes for unsafe languages, expressing end-to-end security guarantees for software components that may

become compromised after encountering undefined behavior–for example, by accessing an array out of bounds.

Our criterion is the first to model dynamic compromise in a system of mutually distrustful components with clearly specified privileges. It articulates how each component should be protected from all the others–in particular, from components that have encountered undefined behavior and become compromised. Each component receives secure compilation guarantees–in particular, its internal invariants are protected from compromised components–up to the point when this component itself becomes compromised, after which we assume an attacker can take complete control and use this component's privileges to attack other components. More precisely, a secure compilation chain must ensure that a dynamically compromised component cannot break the safety properties of the system at the target level any more than an arbitrary attacker-controlled component (with the same interface and privileges, but without undefined behaviors) already could at the source level.

To illustrate the model, we construct a secure compilation chain for a small unsafe language with buffers, procedures, and components, targeting a simple abstract machine with built-in compartmentalization. We give a careful proof (mostly machine-checked in Coq) that this compiler satisfies our secure compilation criterion. Finally, we show that the protection guarantees offered by the compartmentalized abstract machine can be achieved at the machine-code level using either software fault isolation or a tag-based reference monitor.

5.16 Taming Undefined Behavior in LLVM

Chung-Kil Hur (Seoul National University, KR)

Joint work of Chung-Kil Hur, Juneyoung Lee, Yoonseung Kim, Youngju Song, Sanjoy Das, John Regehr, David Majnemer, Nuno P. Lopes
Main reference Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, Nuno P. Lopes: "Taming undefined behavior in LLVM", in Proc. of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017, pp. 633–647, ACM, 2017.
URL http://dx.doi.org/10.1145/3062341.3062343
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.Chung-KilHur.Slides.pdf

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.

5.17 Taming I/O in Intermittent Computing

Limin Jia (Carnegie Mellon University – Pittsburgh, US)

Joint work of Limin Jia, Brandon Lucia, Milijana Surbatovich
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.LiminJia.Slides.pdf

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 unique challenges that existing checkpointing 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.

5.18 Data Refinement for Cogent

Gabriele Keller (The University of New South Wales – Sydney, AU)

Joint work of Gabriele Keller, Christine Rizkallah

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.

5.19 Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"

Vincent Laporte (IMDEA Software Institute – Madrid, ES)

Joint work of Vincent Laporte, Gilles Barthe, Benjamin Grégoire
Main reference Gilles Barthe, Benjamin Grégoire, Vincent Laporte: "Provably secure compilation of side-channel countermeasures", IACR Cryptology ePrint Archive, Vol. 2017, p. 1233, 2017.
URL http://eprint.iacr.org/2017/1233
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.VincentLaporte.Slides.pdf

Software-based countermeasures provide effective mitigation against side-channel attacks, often with minimal efficiency and deployment overheads. Their effectiveness is often amenable to rigorous analysis: specifically, several popular countermeasures can be formalized as information flow policies, and correct implementation of the countermeasures can be verified with state-of-the-art analysis and verification techniques. However, in absence of further justification, the guarantees only hold for the language (source, target, or intermediate representation) on which the analysis is performed.

We consider the problem of preserving side-channel countermeasures by compilation for cryptographic "constant-time," a popular countermeasure against cache-based timing attacks. We present a general method, based on the notion of constant-time-simulation, for proving that a compilation pass preserves the constant-time countermeasure. Using the Coq proof assistant, we verify the correctness of our method and of several representative instantiations.

5.20 The Formal Verification of Compilers and What It Doesn't Say About Security

Xavier Leroy (INRIA – Paris, FR)

Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.XavierLeroy.Slides1.pdf

This talk starts with an overview of the formal verification of compilers, as done in the CompCert and CakeML projects for example.

Verifying the soundness of a compiler means proving that the generated code behaves as prescribed by the semantics of the source program. There are many definitions of interest for "behaves as prescribed." Observational equivalence is appropriate for well-defined source languages such as Java. However, for C and C++, observational equivalence cannot be guaranteed because several evaluation orders are allowed for source programs, while the compiled code implements one of those evaluation orders. Moreover, C and C++ treat run-time errors such as integer division by zero or out-of-bound array accesses as undefined behaviors, meaning that the compiled code is allowed to perform any actions whatsoever, from aborting the program to continuing with random values to opening a security hole.

The CompCert compiler verification project builds on a notion of program refinement that enables the compiler to choose one among several possible evaluation orders, making the program "more deterministic," and also to optimize source-level undefined behaviors away, making the program "more defined." An example of the latter dimension of refinement is the elimination of an integer division `z = x / y` if `z` is unused later: if `y` is 0, the original program exhibits undefined behavior (division by zero), but not the optimized program.

As discussed in the second part of the talk, CompCert-style compiler verification shows the preservation of safety and liveness properties of the source code, but fails to establish the preservation of many security properties of interest. This is illustrated on two examples: constant-time code and unwanted optimizations.

Example 1. Cryptographic code is said to be "constant time" if secret data is never used as argument to conditional branches, memory addressing, or other operations whose execution time depends on the value of the arguments. This "constant time" property does not rule out all side-channel attacks, but avoids the most obvious timing attacks. But is the property preserved by compilation? If the source code is "constant time," is the compiled code "constant time" too? Compilers can destroy the property by introducing conditionals or memory lookups for optimization purposes. This does not invalidate a CompCert-style semantic preservation proof. To reason about constant-time preservation it seems necessary to add observable events for non-constant-time operations to the semantic trace, and reason about the preservation, or removal but not insertion, of such events during compilation.

Example 2. C compilers are allowed to optimize based on the assumption that the source code does not run into undefined behavior. Sometimes, this leads optimizers to amplify a programming error, removing security-relevant checks that follow a possibly-undefined operation. CVE 2009-1879 is an example of such a compiler-amplified security hole. Such misguided optimizations are hard to control because they are close to other desirable optimizations, and both fall out naturally from standard compiler passes such as value analysis and constant propagation. CompCert tries hard to degrade the precision of its value analysis to be conservative with respect to undefined behavior. However, this is a best effort and no formal proof is given that the analysis was degraded enough so that subsequent optimizations preserve security checks.

In conclusion, formal compiler verification in the style of CompCert or CakeML gives many guarantees relevant to safety, but few guarantees relevant to security beyond safety. CompCert tries to handle security code with care, but it's a best effort without confirmation by the proof. More work is needed to semantically characterize the security properties of interest and prove their preservation by compilation.

5.21 Verified Compilation of Noninterference for Shared-Memory Concurrent Programs

Toby Murray (The University of Melbourne, AU)

Joint work of Toby Murray, Robert Sison, Edward Pierzchalski, Christine Rizkallah
Main reference Toby C. Murray, Robert Sison, Edward Pierzchalski, Christine Rizkallah: "Compositional Verification and Refinement of Concurrent Value-Dependent Noninterference", in Proc. of the IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 – July 1, 2016, pp. 417–431, IEEE Computer Society, 2016.
URL http://dx.doi.org/10.1109/CSF.2016.36
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.TobyMurray.Slides.pdf

Shared-memory concurrency is ubiquitous in modern programming, including in security-critical embedded devices. Proofs of information flow control (IFC) for the software that controls such devices have recently become a reality. Yet most of this work to date operates at the level of the small-step semantics for the source programming language. In reality, such programs execute atop a thread scheduler (e.g., the OS kernel), executing binary instructions in fixed slices. We argue that verified noninterference-preserving compilation should be

employed to bridge this semantic gap, and present a theory for compositionally proving preservation of timing-sensitive noninterference for concurrent programs under refinement. We explain how this theory captures the semantics of compiled programs executing under an instruction-based scheduling discipline, and its instantiation in a verified compiler from a simple While language to an idealised RISC language. We report on the current state of this work, which is part of the COVERN project (https://covern.org), and directions for future research.

5.22 Is the Verified CakeML Compiler Secure?

Magnus Myreen (Chalmers University of Technology – Gothenburg, SE)

Joint work of Scott Owens, Yong Kiam Tan, Anthony Fox, Ramana Kumar, Michael Norrish
URL https://cakeml.org/
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.MagnusMyreen.Slides.pdf

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.

5.23 Compiler Optimizations with Retrofitting Transformations: Is There a Semantic Mismatch?

Santosh Nagarakatte (Rutgers University – New Brunswick, US)

Joint work of Santosh Nagarakatte, Jay P. Lim, Vinod Ganapathy
Main reference Jay P. Lim, Vinod Ganapathy, Santosh Nagarakatte: "Compiler Optimizations with Retrofitting Transformations: Is there a Semantic Mismatch?", in Proc. of the 2017 Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2017, Dallas, TX, USA, October 30, 2017, pp. 37–42, ACM, 2017.
URL http://dx.doi.org/10.1145/3139337.3139343
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.SantoshNagarakatte.Slides.pdf

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.

5.24 Plugging Information Leaks Introduced by Compiler Transformations

Kedar Namjoshi (Nokia Bell Labs – Murray Hill, US-NJ)

Joint work of Kedar Namjoshi, Chaoqiang Deng
Main reference Chaoqiang Deng, Kedar S. Namjoshi: "Securing a Compiler Transformation", in Proc. of the Static Analysis – 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings, Lecture Notes in Computer Science, Vol. 9837, pp. 170–188, Springer, 2016.
URL http://dx.doi.org/10.1007/978-3-662-53413-7_9
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.KedarNamjoshi.Slides.pdf

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.

5.25 Relational Logic for Fine-grained Security Policy and Translation Validation

David A. Naumann (Stevens Institute of Technology – Hoboken, US)

Joint work of David A. Naumann, Anindya Banerjee, Mohammed Nikouei
Main reference Anindya Banerjee, David A. Naumann, Mohammad Nikouei: "Relational Logic with Framing and Hypotheses: Technical Report", CoRR, Vol. abs/1611.08992, 2016.
URL http://arxiv.org/abs/1611.08992
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.DavidA.Naumann.Slides.pdf

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.

5.26 Specifications for Dynamic Enforcement of Relational Program Properties

Max S. New (Northeastern University – Boston, US)

Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.MaxS.New.Slides.pdf

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.

5.27 Closure Conversion is Safe-for-Space

Zoe Paraskevopoulou (Princeton University, US)

Joint work of Zoe Paraskevopoulou, Andrew Appel

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.

5.28 Linking Types: Bringing Fully Abstract Compilers and Flexible Linking Together

Daniel Patterson (Northeastern University – Boston, US)

Joint work of Daniel Patterson, Amal Ahmed
Main reference Daniel Patterson, Amal Ahmed: "Linking Types for Multi-Language Software: Have Your Cake and Eat It Too", in Proc. of the 2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA, LIPIcs, Vol. 71, pp. 12:1–12:15, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2017.
URL http://dx.doi.org/10.4230/LIPIcs.SNAPL.2017.12
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.DanielPatterson.Slides.pdf

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.

5.29 A Project on Secure Compilation in the Context of the Internet of Things

Tamara Rezk (INRIA – Sophia Antipolis, FR)

Joint work of Tamara Rezk, Frédéric Besson, Thomas Jensen, Alan Schmitt, Gérard Berry, Nataliia Bielova, Ilaria Castellani, Manuel Serrano, Claude Castelluccia, Daniel Le Métayer
URL http://cisc.gforge.inria.fr/

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.

5.30 A Formal Equational Theory for Call-By-Push-Value

Christine Rizkallah (The University of New South Wales – Sydney, AU)

Joint work of Christine Rizkallah, Dmitri Garbuzov, Steve Zdancewic
URL https://github.com/secure-compilation/ds-2018/raw/master/18201.ChristineRizkallah.Preprint.pdf
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.ChristineRizkallah.Slides1.pdf

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.

5.31 Secure Compilation–Understanding the Endpoints

Peter Sewell (University of Cambridge, GB)

Joint work of many people
URL http://www.cl.cam.ac.uk/users/pes20/rems
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.PeterSewell.Slides.pdf

In this talk I described ongoing work in the REMS and CHERI projects to define the architecture and C-language abstractions, both for current mainstream architectures (especially ARMv8-A and RISC-V, with some work also for IBM POWER and x86) and mainstream ISO / de facto C, and for the research CHERI architecture and CHERI C language. I also described work on WebAssembly semantics.

5.32 Constant-Time Crypto Programming with FaCT

Deian Stefan (University of California, San Diego, US)

Joint work of Deian Stefan, Fraser Brown, Sunjay Cauligi, Ranjit Jhala, Brian Johannesmeyer, John Renner, Gary Soeller, Riad Wahby, Conrad Watt
Main reference Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, Deian Stefan: "FaCT: A Flexible, Constant-Time Programming Language", in Proc. of the IEEE Cybersecurity Development, SecDev 2017, Cambridge, MA, USA, September 24-26, 2017, pp. 69–76, IEEE, 2017.
URL http://dx.doi.org/10.1109/SecDev.2017.24
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.DeianStefan.Slides.pdf

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).

5.33 C-Level Tag-Based Security Monitoring

Andrew Tolmach (Portland State University, US)

Joint work of Andrew Tolmach, Sean Anderson, Catalin Hritcu, Benjamin C. Pierce
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.AndrewTolmach.Slides.pdf

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.

5.34 Verifying the Glasgow Haskell Compiler Core Language

Stephanie Weirich (University of Pennsylvania – Philadelphia, US)

Joint work of Stephanie Weirich, Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.StephanieWeirich.Slides.pdf

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?

5.35 Vellvm: Verifying the LLVM

Steve Zdancewic (University of Pennsylvania – Philadelphia, US)

Joint work of Steve Zdancewic, Dmitri Garbuzov, William Mansky, Christine Rizkallah, Yannick Zakowski
Slides https://github.com/secure-compilation/ds-2018/raw/master/18201.SteveZdancewic.Slides.pdf

I will describe the Vellvm project, which seeks to provide a formal framework for developing machine-checkable proofs about LLVM IR programs and translation passes. I'll highlight some of the "good," the "bad" and the "ugly" things about our prior LLVM developments, which motivates our ongoing work to re-engineer the Coq formalization.

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.

Participants

- Amal Ahmed
Northeastern University – Boston, US
- Gilles Barthe
IMDEA Software – Madrid, ES
- Nick Benton
Facebook – London, GB
- Frédéric Besson
INRIA – Rennes, FR
- Pramod Bhatotia
University of Edinburgh, GB
- Lars Birkedal
Aarhus University, DK
- Roberto Blanco
INRIA – Paris, FR
- William Bowman
Northeastern University – Boston, US
- Stefan Brunthaler
Universität der Bundeswehr – München, DE
- David Chisnall
University of Cambridge, GB
- John T. Criswell
University of Rochester, US
- Dominique Devriese
KU Leuven, BE
- Derek Dreyer
MPI-SWS – Saarbrücken, DE
- Akram El-Korashy
MPI-SWS – Saarbrücken, DE
- Cédric Fournet
Microsoft Research UK – Cambridge, GB
- Deepak Garg
MPI-SWS – Saarbrücken, DE
- Chris Hawblitzel
Microsoft Research – Redmond, US
- Catalin Hritcu
INRIA – Paris, FR
- Chung-Kil Hur
Seoul National University, KR
- Limin Jia
Carnegie Mellon University – Pittsburgh, US
- Gabriele Keller
UNSW – Sydney, AU
- Vincent Laporte
IMDEA Software – Madrid, ES
- Xavier Leroy
INRIA – Paris, FR
- Toby Murray
The University of Melbourne, AU
- Magnus Myreen
Chalmers University of Technology – Göteborg, SE
- Santosh Nagarakatte
Rutgers University – Piscataway, US
- Kedar Namjoshi
Nokia Bell Labs – Murray Hill, US
- David A. Naumann
Stevens Institute of Technology – Hoboken, US
- Max S. New
Northeastern University – Boston, US
- Scott Owens
University of Kent – Canterbury, GB
- Zoe Paraskevopoulou
Princeton University, US
- Marco Patrignani
Universität des Saarlandes, DE
- Daniel Patterson
Northeastern University – Boston, US
- Frank Piessens
KU Leuven, BE
- Tamara Rezk
INRIA Sophia Antipolis, FR
- Christine Rizkallah
UNSW – Sydney, AU
- Peter Sewell
University of Cambridge, GB
- Deian Stefan
University of California – San Diego, US
- Andrew Tolmach
Portland State University, US
- Stelios Tsampas
KU Leuven, BE
- Neline van Ginkel
KU Leuven, BE
- Stephanie Weirich
University of Pennsylvania – Philadelphia, US
- Steve Zdancewic
University of Pennsylvania – Philadelphia, US