

Secure Compilation

Edited by

David Chisnall¹, Deepak Garg², Catalin Hritcu³, Mathias Payer⁴

1 Microsoft Research – Cambridge, UK, david.chisnall@microsoft.com

2 MPI-SWS – Saarbrücken, DE, dg@mpi-sws.org

3 MPI-SP – Bochum, DE, catalin.hritcu@mpi-sp.org

4 EPFL – Lausanne, CH, mathias.payer@nebelwelt.net

Abstract

Secure compilation is an emerging field that puts together advances in security, programming languages, compilers, verification, systems, and hardware architectures in order to devise more secure compilation chains that eliminate many of today’s security vulnerabilities and that allow sound reasoning about security properties in the source language. 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 against linked low-level code, which can call and return to arbitrary instructions or smash the stack, blatantly violating the high-level abstraction. Other problems arise because today’s languages fail to specify security policies, such as data confidentiality, and the compilation chains thus fail to enforce them, especially against powerful side-channel attacks. The emerging secure compilation community aims to address such problems by identifying precise security goals and attacker models, designing more secure languages, devising efficient enforcement and mitigation mechanisms, and developing effective verification techniques for secure compilation chains.

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 designing security enforcement and attack-mitigation mechanisms in both software and hardware, and on developing formal verification techniques for secure compilation.

Seminar November 28–December 3, 2021 – <http://www.dagstuhl.de/21481>

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.11.10.173

Edited in cooperation with Roberto Blanco (MPI-SP – Bochum, DE)



Except where otherwise noted, content of this report is licensed under a Creative Commons BY 4.0 International license

Secure Compilation, *Dagstuhl Reports*, Vol. 11, Issue 10, pp. 173–204

Editors: David Chisnall, Deepak Garg, Catalin Hritcu, and Mathias Payer



DAGSTUHL
REPORTS

Dagstuhl Reports
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


1 Executive Summary

David Chisnall (Microsoft Research – Cambridge, UK)

Deepak Garg (MPI-SWS – Saarbrücken, DE)

Catalin Hritcu (MPI-SP – Bochum, DE)

Mathias Payer (EPFL – Lausanne, CH)

License  Creative Commons BY 4.0 International license
© David Chisnall, Deepak Garg, Catalin Hritcu, and Mathias Payer

Secure compilation is an emerging field that puts together advances in security, programming languages, compilers, systems, verification, and hardware architectures to devise compilation chains that eliminate security vulnerabilities, and allow sound reasoning about security properties in the source language. For example, all modern languages define valid control flows, e.g., calls must always return to the instruction after the calling point, and many security-critical analyses such as data flow analysis rely on programs adhering to these valid control flows. However, today’s compilation chains (compilers, linkers, loaders, runtime systems, hardware) cannot efficiently prevent violations of source-level control flows by co-linked low-level code, which can call and return to arbitrary instructions or smash the stack, blatantly violating the high-level abstraction. Other problems arise because languages fail to specify security policies, such as data confidentiality, and the compilation chains thus fail to enforce them, especially against powerful attacks such as those based on side channels. Yet other problems arise because enforcing source-level abstractions requires runtime checks with noticeable overhead, so compilation chains often forego security properties in favor of efficient code. The emerging field of secure compilation aims to address such problems by:

1. **Identifying precise security goals and attacker models.**

Since there are many interesting security goals and many different kind of attacks to defend against, secure compilation is very diverse. Secure compilation chains may focus on providing (some degree of) type and memory safety for unsafe low-level languages like C and C++, or on providing mitigations that make exploiting security vulnerabilities more difficult. Other secure compilation chains use compartmentalization to limit the damage of an attack to only those components that encounter undefined behavior, or to enforce secure interoperability between code written in a safer language (like Java, C#, ML, Haskell, or Rust) and the malicious or compromised code it links against. Yet another kind of secure compilation tries to ensure that compilation and execution on a real machine does not introduce side-channel attacks.

2. **Designing secure languages.**

Better designed programming languages and new language features can enable secure compilation in various ways. New languages can provide safer semantics, and updates to the semantics of old unsafe languages can turn some undefined behaviors into guaranteed errors. Components or modules in the source language can be used as units of compartmentalization in the compilation chain. The source language can also make it easier to specify the intended security properties. For instance, explicitly annotating secret data that external observers or other components should not be able to obtain (maybe indirectly through side channels) may give the compilation chain the freedom to more efficiently handle any data that it can deduce is not influenced by secrets.

3. **Devising efficient enforcement and mitigation mechanisms.**

An important reason for the insecurity of today’s compilation chains is that enforcing security can incur prohibitive overhead or significant compatibility issues. To overcome these problems, the secure compilation community is investigating various efficient security

enforcement mechanisms such as statically checking low-level code, compiler optimizations, software rewriting (e.g. software fault isolation), dynamic monitoring, and randomization. Another key enabler is the emergence of new hardware features that enable efficient security enforcement: access checks on pointer dereferencing (e.g. Intel MPX, Hardbound, WatchdogLite, Oracle SSM, SPARC ADI, or HWASAN), protected enclaves (e.g. Intel SGX, ARM TrustZone, Sanctum, or Sancus), capability machines (e.g. CHERI, Arm Morello), or micro-policy machines (e.g. Draper PUMP, Dover CoreGuard). The question is how such features can enable various security features in source languages efficiently, i.e., how hardware extensions can provide enforcement mechanisms for security properties.

4. **Developing effective verification techniques for secure compilation chains.**

Criteria for secure compilation are generally harder to prove than compiler correctness. As an example, showing full abstraction, a common criterion for secure compilation, requires translating any low-level context attacking the compiled code to an equivalent high-level context that can attack the original source code. Another example is preservation of secret independent timing even in the presence of side-channels, as required for “constant-time” cryptographic implementations, which can require more complex simulation proofs than for compiler correctness. Finally, scaling such proofs up to even a simple compilation chain for a realistic language is a serious challenge that requires serious proof engineering in a proof assistant.

The Secure Compilation Dagstuhl Seminar 21481 attracted a large number of excellent researchers with diverse backgrounds. The 42 participants (12 on site, 30 remote) represented the programming languages, formal verification, compilers, security, systems, and hardware communities, which led to many interesting points of view and enriching discussions. Due to COVID-19 pandemic-related travel restrictions and uncertainties, many of the participants had to participate remotely using a combination of video conferencing, instant messaging, and ad-hoc gatherings. Despite this mixed environment, discussions thrived. Some of these conversations were ignited by the 5 plenary discussions and the 28 talks contributed by the participants. The contributed talks spanned a very broad range of topics: formalizing ISA security guarantees, hardware-software contracts, detection and mitigation of (micro-architectural) side-channel attacks, securing trusted execution environments, memory safety, hardware-assisted testing, sampled bug detection, formal verification techniques for low-level languages and secure compilation chains, machine-checked proofs, stack safety, integrating hardware-safety guarantees, effective compartmentalization and its enforcement, cross-language attacks, security challenges of software supply chains, capability machines, (over-)aggressive compiler optimizations, concurrency, new programming language abstractions, compositional correct/secure compilation, component safety, compositional verification, contextual and secure refinement, hardening WebAssembly, secure interoperability, (not) forking compilers, interrupts, hardware design, and many more. 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 second edition and the positive feedback received afterwards, we believe that this Dagstuhl Seminar should be repeated in the future, when hopefully all the participants will be able to attend onsite. One important aspect that could still be improved in future editions is spurring more participation from the systems and hardware communities, especially people working at the intersection of these areas and security or formal verification.

2 Table of Contents

Executive Summary

David Chisnall, Deepak Garg, Catalin Hritcu, and Mathias Payer 174

Plenary Discussions

Real-world deployment and remaining frontiers for secure compilation research
Mathias Payer 178

Microarchitectural and side-channel attacks
Marco Guarnieri 179

Designing New Security Architectures and Verifying their Properties
Shweta Shinde 181

Verification techniques for secure compilation
Dominique Devriese 185

Secure interoperability and compartmentalization
David Chisnall 186

Overview of Talks

Enforcement and compiler preservation of fine-grained constant-time policies
Gilles Barthe 190

Formalizing Stack Safety as a Security Property
Roberto Blanco 190

Are Compiler Optimizations Doing it Wrong? An Investigation of Array Bounds
Checking Elimination
Stefan Brunthaler 191

Cross-Language Attacks
Nathan Burrow 191

Securing Interruptible Enclaved Execution on Small Microprocessors
Matteo Busi 192

Project Verona: An abstract machine allowing partial verification
David Chisnall 192

On information flow preserving refinement
Mads Dam 193

Formalizing ISA security guarantees in the form of universal contracts
Dominique Devriese 193

Proof techniques for secure compilation with memory sharing
Akram El-Korashy 194

Preserving Memory Safety from C to MSWasm
Anitha Gollamudi 194


Contract-aware secure compilation: a foundation for side-channel resistant compilers
– Challenges and open questions
Marco Guarnieri 195

Formally verifying a secure compilation chain for unsafe C components <i>Catalin Hritcu</i>	195
Conditional Contextual Refinement <i>Chung-Kil Hur</i>	196
CompCertO: Compiling Certified Open C Components <i>J�r�mie Koenig</i>	196
Changing Compilation without Changing the Compiler <i>Per Larsen</i>	197
WebAssembly as an intermediate language for safe interoperability <i>Zoe Paraskevopoulou</i>	197
Compositional Secure Compilation against Spectre <i>Marco Patrignani</i>	197
Automatic inference of effective compartmentalization policies <i>Mathias Payer</i>	198
Hardware-Software Contracts and Secure Programming <i>Jan Reineke</i>	199
Hardware-assisted testing in production <i>Kostya Serebriany</i>	199
Morello status and verification <i>Peter Sewell</i>	200
A Wishlist for the Next Generation of Trusted Execution Environments <i>Shweta Shinde</i>	200
Swivel: Hardening WebAssembly against Spectre <i>Deian Stefan</i>	200
Compiler-based Side Channel Detection and Mitigation <i>Gang Tan</i>	201
Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code <i>Thomas Van Strydonck</i>	201
Software Supply Chains: Challenges and Opportunities <i>Nikos Vasilakis</i>	201
How we design hardware and what is costs? <i>Ingrid Verbauwhede</i>	202
Verifying Speculation Security of Processor Implementations <i>Drew Zagieboylu</i>	202
Participants	203
Remote Participants	203

3 Plenary Discussions

3.1 Real-world deployment and remaining frontiers for secure compilation research

Discussion led by Mathias Payer (EPFL – Lausanne, CH)

License  Creative Commons BY 4.0 International license
© Mathias Payer

In this session we focused on two key topics: real-world deployment and remaining frontiers for secure compilation research. Both topics are challenging, the former focusing on how we can introduce formal methods into the compilation toolchain, making developers more aware of the different advantages, lowering the barrier to entry for using our tools, and addressing practical deployment concerns. The latter focuses on where to go next such as targeting different compilers, better SAT/SMT solving, scalability issues, targeting large code bases, as well as combining formal methods with other techniques.

3.1.1 Topic 1: deployment

We started the discussion by focusing on issues that keep secure compilation techniques from being applied in practice. The discussion focused around software testing but mostly focused on mitigations. Mitigations are defenses that make exploitation of remaining bugs harder. There is an inherent trade-off between the incurred overhead and the effectiveness of the mitigations. Generally speaking, today it is extremely challenging to deploy new mitigations in practice.

Maintaining mitigations comes inherently at higher overheads on engineering and performance. Any mitigation that is added becomes part of the TCB. After being deployed, mitigations will have to be maintained and will increase engineering complexity by, e.g., making debugging and development more challenging. We don't understand the process of how mitigations transition into practice fully yet but they are often implemented by major players such as Linux kernel developers and companies like Microsoft and Google. A recent example of a mitigation that was changed (apart from CFI being deployed broadly) are stack canaries slowly being deprecated in favor of shadow stacks on the Android operating system. This process is showing how challenging sunsetting mitigations actually is.

The true cost of security is not just the overhead of mitigations or the cost of patching but the real cost is the downtime of important systems. For confidentiality (in addition to integrity), this is a big issue, so improvements will be extremely important. Confidentiality attacks can be passive, making them harder and slower to find. The Heartbleed bug was one such confidentiality attack. On one hand, we looked at network captures from the past and did not find any exploitation that happened. On the other hand, people did not update their keys after compromise or updated them wrongly (with weaker keys).

As a community, we need to go beyond either or approaches and work on finding metrics to evaluate the benefits of a mitigation along with coming up with models on how to maintain them as well as deploying them in a hybrid manner to anticipate how they will be sunsetted whenever a stronger and better mitigation comes around (or a bug class disappears).

3.1.2 Topic 2: research frontiers

After extensively talking about deployment problems we started discussing research frontiers on where and how secure compilation can help. The first seminar on secure compilation was

right after the disclosure of Spectre and Meltdown. This resulted in speculative execution being a major ad-hoc topic and, three years later it has evolved into a key focus of secure compilation, with a large number of researchers.

The underlying issue is that hardware may not always do what it is supposed to do and even though it gives us guarantees, these guarantees may not hold. One option is to move towards fault tolerant computation where we adjust our assumptions in our models that the hardware may fail. Including this assumption will allow us to give guarantees despite hardware failures. When looking at the different stacks, so far security is dependent on each individual layer without cross-cutting concerns. As a way forward, we need to drive the argument across the different layers and handle security between layers of abstractions.

In addition to side channels, we also looked at integrity violations such as RowHammer. In the past, hardware was modeled for performance. Architects exclusively optimized for performance at lower cost. Without clear benefits, users are unwilling to pay for security, we need to justify why and how this is necessary. Focusing on fault tolerance, redundancy could solve the issue at some constant cost factor. The underlying challenge is what the cost of mitigating the issue would be at the hardware level – is it really 2x or could it be lower? As a follow up question, we wondered if and how we can add security to our performance model (i.e., not just focusing on throughput, latency, power). So far security is neither present nor added and this needs to change.

Another research frontier is specifications at all levels of abstractions. While ARM released some specifications, they can only partially be turned into guarantees and not all of them are in a directly usable state. Evidently, people are not very good at writing specifications (or even writing functional tests). We wondered if we need weaker specifications that nevertheless remain useful for verification or better composition techniques. Deriving better and tighter/more precise specifications will be an interesting research frontier, especially when moving towards side channels and hardware faults.

3.2 Microarchitectural and side-channel attacks

Discussion led by Marco Guarnieri (IMDEA Software – Madrid, ES)

License  Creative Commons BY 4.0 International license
© Marco Guarnieri

In this session, we focused on microarchitectural and side-channel attacks. We started by discussing “How can we design principled countermeasures and mitigations against these attacks?” The discussion on this point highlighted that a clear attacker model and a precise description of the security-relevant hardware/software interface are needed to design principled mitigations. Next, we discussed the hardware/software interface for security and, in particular, the security guarantees that hardware should provide to software and how to express them. We concluded by discussing whether secure compilation techniques can help in securing the hardware/software interface. We now present a short summary of each discussion point.

3.2.1 How can we design principled countermeasures and mitigations?

We started the discussion by observing that software has only limited control on microarchitectural aspects and side-effects (e.g., using memory fences to limit the scope of speculation or dedicated commands to flush internal caches and buffers). Even with these commands, however, it is often difficult to build *principled* mitigations due to a lack of detailed microarchitectural models.

One possible way forward consists in extending the ISA with dedicated commands for controlling part of the microarchitectural state. The hardware implementation, then, will be in charge of correctly implementing these commands in a secure manner. We discussed two alternative approaches for this:

- A coarse-grained approach where the processor provides two execution modes: a secure execution mode and an insecure execution mode. The former provides strong security against side-channel and microarchitectural attacks (e.g., by disabling several processor optimizations and reducing resource sharing) at the price of a performance overhead, whereas the latter provides no security guarantees. In this case, programmers need to precisely identify how to partition programs into a secure and insecure parts.
- A fine-grained approach where programmers can specify isolation programmatically down to the microarchitectural level. In principle, with more control on the microarchitectural state, programmers could implement secure code with better performance. Secure compilers could also help in correctly enforcing the desired security properties. Programmatic partitioning at microarchitectural level might also have performance benefits. However, properly using microarchitectural partitioning at software-level might be very challenging (opening the door to potential vulnerabilities).

Another important aspect in mitigating microarchitectural attacks is that countermeasures often come with some performance overhead. As a result, designers need to consider the trade-off between security and performance. For this, we need quantitative measures for security. For instance, in hardware power-based side-channel attacks, security is quantified using the the number of samples needed to obtain some information, with higher numbers being better for security. Work on quantitative information flow can provide the theoretical foundations for building these microarchitectural quantitative measures.

3.2.2 What security guarantees should hardware provide to software?

Most of the recent microarchitectural attacks break the *intuitive* assumptions about the security guarantees that hardware should provide to software. For instance, Spectre attacks break the assumption that code is executed following a program’s control flow, whereas Rowhammer attacks break assumptions about memory integrity.

A precise specification about the security guarantees that hardware provides to software is needed as a starting point for building secure systems. Such a specification establishes a *contract* between hardware and software.

Recently, there have been several proposals for formalizing such hardware/software contracts. We discussed several aspects of these specifications:

- Existing proposals are rather narrow and they mostly focus on timing-based attacks. It is unclear whether and how these proposals can be extended to other classes of attacks such as power-based side-channel attacks.
- A key point in defining such a specification is identifying the right level of abstraction for microarchitectural components and side-effects. A contract that comes with a detailed microarchitectural model imposes more constraints on hardware designers. At the same time, a more detailed contract can provide information, e.g., a specific cache replacement policy, that programmers can use to build systems with better performance.
- Such specifications need to be easy to use by software-level tools like program analyses and compilers.
- These hardware/software security specifications need to distribute “security obligations” between hardware and software. Additionally, we need ways to change such specifications as hardware and software systems evolve.

3.2.3 How can secure compilation help to secure the hardware/software interface?

We discussed several ways in which secure compilation could help in securing the hardware/software interface.

From a foundational perspective, secure compilation criteria like full abstraction and preservation of specific properties (such as non-interference) can provide inspiration for the formalization of hardware/software contracts for microarchitectural security.

From a practical perspective, compilers can assist in building secure systems since they can inform the hardware about which information is sensitive and should be protected. For instance, in the coarse-grained model mentioned before compilers can help in partitioning the code, whereas in the fine-grained model compilers can help in correctly inserting instructions for partitioning the microarchitectural state.

3.3 Designing New Security Architectures and Verifying their Properties

Discussion led by Shweta Shinde (ETH Zürich, CH)

License © Creative Commons BY 4.0 International license
© Shweta Shinde

We have seen a rise in academic and industry-led efforts for building new architectures. These advances were motivated by several reasons such as diminishing returns of Moore's law, emphasis on energy-efficient designs, and opportunities presented by unified memory architectures. In addition to performance improvements, this shift has led to an opening where security-centric thinking can either be tightly coupled or orthogonality added to these new architectures. In this session, we discussed the ramifications of this shift, particularly to assess the possibility of building secure designs with strong security guarantees that are amenable to formal verification.

3.3.1 Are security architectures actually on the rise, and why?

The discussion started by questioning the veracity of the claim that security architectures are on the rise and within the scope of real-world deployments. The hardware design and verification flows have adopted agile deployment pipelines. This allows researchers to come up with new designs and prototype them quickly. First, there are extensions (e.g., CHERI) that can be tested out quickly because they do not disrupt the rest of the flow (e.g., perform operations only while accessing memory). Second, several primitives can be applied by raising the ISA abstraction and adding new instructions without changing the critical parts of the architecture. Such proposals offer easy adoption paths for manufacturers, who are willing to enable security features.

When it comes to deciding if some primitives are inherently suited for hardware or software, there is a risk of pushing the responsibility. In this case, an extreme example, can be viewing the hardware only as a way to achieve high performance; whereas functional security and isolation are solely a responsibility of software. A counter view is to treat hardware as a component that provides a common abstraction such that the functionality can be appropriately leveraged in software. If the software layer has to simulate a different abstraction because the underlying hardware does not natively support it, this can imply that there is indeed a semantic gap that needs to be bridged with better abstractions. The speculative side-channel attacks are a good example of such a mismatch.

3.3.2 Need for First-class Security Primitives

There are two aspects when it comes to designing robust and effective security primitives. First, one can be ambitious and provide elaborate primitives that are indeed useful to solve large classes of attacks. However, if they are not easy to use, adapt, and apply then such primitives are not immediately practical (e.g., homomorphic encryption, oblivious RAM). Second, given an intuitive and useful primitive, should it be implemented and enforced in hardware or software. This is a question of optimizing non-security aspects such as performance and compatibility. So, the answer usually depends on the specific primitive. For example, doing physical memory isolation can be easier at the hardware level because it is simple enough. However, ensuring non-interference might be challenging to achieve purely in hardware.

3.3.3 Risks and Benefits of Adding New Hardware Primitives

It is crucial to be selective with the hardware primitives. Otherwise, we run a risk of having too many primitives in hardware. Such a proliferation in the best case causes fragmentation and in the worst case can lead to poor security of the overall system. Even if a primitive can be implemented in hardware, practical limitations impose several constraints (e.g., power, area, memory latency). Further, hardware development cycles are much longer. A rough estimate is 5 years for prototyping one generation; at least 2-3 generations before the performance characteristics of a new primitive are acceptable and up to the designers' expectations. Lastly, removing features from hardware is non-trivial and expensive. It breaks compatibility, incurs large changes to software and tools, and may severely annoy customers who are vested in using the hardware features. On the upside, hardware has relatively fewer complex interactions and is a good vantage point for certain enforcement. Thus, at least for simple primitives, implementing them in hardware gives one a better chance of getting it right with a high potential impact. To bring in the best of both worlds, the optimal strategy might be to let the hardware provide only bare minimum security functionality. The software can handle the complex aspects with the potential of several cheap design iterations. The only downside of this approach is that one has to then trust the software to use the primitives correctly and guarantee the overall security without leaving gaps that the attacker can exploit. Perhaps, this is where secure compilation can be vital.

3.3.4 Secure Compilation

In the case of secure compilation, one always considers that the attacker operates at the lowest level. But is this a realistic assumption, and if so, is it practical to protect against such an adversary? For example, researchers have demonstrated the practicality of severe attacks (e.g., cache side-channels, fault injection) that are not addressable purely in software. Unfortunately, the layers at which fixes and mitigations could be introduced (i.e., hardware design) are very removed from the layers that suffer directly from poor security (i.e., software). Moreover, in practice, one has to consider what threats are important to the end-user. If the customers express a strong desire for a given feature, then designers can carve a way forward to an industrial evaluation.

3.3.5 Practical Roadblocks for Hardware-enforced Security

Besides the economic incentive, there are several reasons for investigating hardware features for strengthening security. If a small slowdown in hardware could result in speedups in software, such a slowdown would still be beneficial. However, there are several barriers

to making these strides. First, quantifying and associating performance gains to design decisions is not straightforward, especially in complex architectures. Quantifying the costs of a given overhead, not only directly, but how they propagate across pipelines and scale to large numbers of computers is non-trivial. Without concrete and clear root-cause analysis, it is difficult to convince the importance of such changes to the stakeholders. There is no immediate clear path forward from these challenges.

Many performance studies traditionally start with simulations. For example, academic researchers might first build and prototype the smallest steps to validate their ideas and then rely on industry partners to build complete products. Despite the appeal of this approach, the error margin of simulator studies is large enough that statistical noise is in the same order of magnitude, or bigger, than the effects that one intends to measure. Conversely, reasonably accurate simulators are woefully slow. Thus, meaningfully extrapolating insights from prototypes is challenging. In this, as in many other matters, it is clear that early collaborations between academia and industry is crucial.

There are a few lessons that we have learned by building multiple hardware primitives. First, incentives play an important role in the practical adoption of hardware. A clear and drastic reduction in security risks is still a good incentive. However, OS and mobile systems developers do not wish to take responsibility for hardware problems. If we identify a security mechanism or part thereof, that needs to be implemented in hardware, we have to consider several constraints. These include performance, throughput, area, power, and energy. Estimating the exact impact of a mechanism of these factors is an open problem. However, collaborations can help identify non-negotiable constraints early on.

3.3.6 Role of Formal Verification

Formal verification has been traditionally used in security settings to strengthen reasoning. However, verification is also useful for the functional validation of hardware features and reliably projecting costs such as area and timing delays. If one applies rigorous formal modeling, it can open up avenues for simplifying the designs and to make them easy for verification. In addition, such efforts pay off even in the hardware verification phase. There have been several success stories of such a cycle, including virtual memory, protection bits, capability machines, and more recently CHERI. A lot of these success stories have roots in early hardware abstractions for scalability. However, in the last decade or so, there has been a steady rise in the number of purely security-centric hardware extensions. Other than the publicly available extensions that have seen adoption, manufacturers are more receptive to such ideas. It is worth noting that recent extensions are driven by software – techniques that have stood the test of time in software and compiler enforced security are now being moved to hardware for efficiency (e.g., pointer authentication). Moving forward, there are a few key principles that can help accelerate such adoption success. Designers can start with clear specifications and then build the primitives. This opens up an easy path for ISA-level verification at a later stage. It further allows us to cross-check if the hardware implementation indeed adheres to the specification.

3.3.7 Potential Avenues for New Security Primitives

Given the wide-scale hardware support for trusted execution environments (e.g., Intel SGX) and the success of non-TEE primitives such as CHERI, is there any scope for building on these wins? For example, could one re-purpose CHERI to support TEE primitives? If so, would it address problems beyond the scope of TEEs (e.g., side channels)? One has to

be careful when it comes to confidentiality, since it requires addressing all possible side-channel attacks. Tangentially, modern cloud deployments use virtual machine and container abstractions. Is there potential for new primitives that are closer to and well-tailored for these abstractions? We have seen that hardware designers are keen on supporting fast virtualization (e.g., Intel VT, ARM CCA) as well as language-specific extensions (e.g., for JavaScript). Are there low-hanging fruits that are within the scope of hardware adoption either based on programming models (e.g., secure language virtual machines) or system abstractions (e.g., library/object isolation)? This requires considerations about developer efforts and ease of using the abstractions correctly. The interface exposed to the developers should not be drastically in contrast to the traditional programming model and should be easy to infer and/or encapsulate at the programming abstraction level (e.g., libraries).

3.3.8 Ramifications for Verification Efforts

Intuitively, adding clean abstractions in hardware does help in overall reasoning. For example, introducing explicit hardware-software contracts allows one to reason about specific types of speculative side-channel attacks. On the other hand, the added complexity of the ISA may impact the proof efforts. For example, in proofs of compartmentalization, the attacker model is arbitrary assembly code. Thus, the security proof has to reason about all possible instructions and their side effects. There are well-known mechanisms to circumvent these challenges. To scale security proofs, designers use simplified models (e.g., infinite memory, unbound number of enclaves). However, one has to be cautious in assessing and closing any gaps between model and reality. One way to do this is by improving the model and checking the assumptions. We have seen that these efforts do pay off (e.g., static analyses are now applied to increasingly larger systems). On the other extreme, it has been shown that substantive formalization and sufficient efforts of end-to-end systems are feasible (e.g., the verified light bulb). With regards to prospects of hardware primitives, free and open-source models (e.g., RISC-V) gives researchers an advantage to easily explore design options without the need to circumvent complex legacy systems or wait for hardware manufacturers to adopt the designs. One opportunity is to build secure compilers from the ground up, in a similar spirit to CompCert, but now for hardware.

3.3.9 Summary

There have been several success stories of introducing new hardware primitives that either indirectly aid or are directly beneficial for improving security. Although this requires substantial efforts from both industry and academia, the overall barrier to entry has and continues to be reduced. The next step toward sustainable security is to make formal verification an integral part of the process. Looking at the evolution of adopting security as a first-class concern, we anticipate a similar journey for formal verification.

3.4 Verification techniques for secure compilation

Discussion led by Dominique Devriese (KU Leuven, BE)

License © Creative Commons BY 4.0 International license
© Dominique Devriese

The discussion was centered on three major themes:

- (1) what to prove about a secure compiler,
- (2) how to prove secure compilation properties, and
- (3) machine-checked proofs of secure compilation.

3.4.1 What to prove about a secure compiler?

There are generally two types of properties that one can formally prove about a secure compiler. First: robust preservation of contextual equivalence (full abstraction) or the generalization to robust preservation of hyperproperties on traces of interaction with the outside world. These are robust in the sense that they consider security properties that hold in the presence of an active attacker represented by a program context. Second: in some settings, secure compilation is proven in a non-robust form as preservation of properties on a trace. In such properties, a passive attacker is considered that only interacts with a program through observing or providing inputs on the trace. This is the case, for example, for constant-time preservation in constant-time compcert and the Jasmin compiler. Generally, there is a consensus that it is good that many properties have been proposed so that we can choose the best suited property for a particular system.

It was pointed out that some properties take a kind of hybrid perspective, combining both a passive and active attacker. This is the case, for example, for the typical interpretation of robust non-interference preservation, where the attacker is present as a context, but his/her goal is to learn secrets from the trace of interactions with the outside world. It is not clear whether the attacker should be able to manipulate traces in this model and how secrets may enter a system. Some people think only the context should represent the active attacker and traces and hyperproperties should represent only a success criterion for the attacker. It is not generally clear how contexts (active attackers) can be given all the capabilities of traces.

3.4.2 How to prove secure compilation properties?

We have talked about the fact that weaker secure compilation properties can be easier to prove. For example, back-translations may depend on more information, when proving robust preservation of more restricted classes of hyperproperties. This has been used in some results, but how large the advantage is, can depend on the compiler at hand. For preservation of non-robust properties, one can do without a back-translation. For example, constant-time preservation can be proven using specific forms of simulation cubes and related techniques. Overapproximating an intended security property can also help to simplify proofs (for example a policy on accessing information as an overapproximation for information flow).

A general challenge is the reuse of existing proofs when proving secure compilation. We would like to be able to construct secure compilation proofs in such a way that they can be reused in compiler chains. However, there is also interest in reusing a compiler correctness proof when reasoning about a secure compiler. Some people point out that it can be beneficial to decompose secure compiler (passes) into several smaller secure compiler passes. Combining the results of secure compilation passes may not be obvious when they prove (robust) preservation of different (hyper)properties. A framework like that of CompCertM

may help to formulate intended secure compilation results in the presence of different calling conventions. In addition to vertical composability, horizontal composability is an important challenge, where it is not yet entirely clear how this should work.

3.4.3 Machine-checked secure compilation proofs

The final topic of discussion was the machine verification of secure compilation proofs. There are very few papers that go all the way through this. These proofs are inherently hard and it simply follows that mechanizing them is difficult.

It was asked whether sufficient value is attached to machine-verified proofs. Generally, machine verification is regarded as an important extra quality for a paper (or an extra contribution in a journal version), but not a result in itself, even though the amount of effort can be similar to the amount of effort for the paper itself. A mechanization can sometimes simply increase confidence in a result without adding much extra insights, but in some cases, they have been known to uncover important problems that require additional insights to solve.

Given the difficulty of machine-verified secure compilation proofs, some people suggest that we should attempt to build libraries of reusable components and proofs. In the verified compilation community, we have recently started seeing many results building on and improving existing large formalizations (particularly CompCert) rather than starting from scratch. It is not clear whether the field of secure compilation is already sufficiently mature for a similar evolution to take place. To facilitate this further, it would be good if we have reusable proofs of security primitives, common languages to express properties and traces, shared languages for interacting with the outside world, contracts for side-channel leakage etc.

3.5 Secure interoperability and compartmentalization

Discussion led by David Chisnall (Microsoft Research – Cambridge, UK)

License  Creative Commons BY 4.0 International license
© David Chisnall

Modern programs are typically written in more than one language. There is a growing trend towards writing as much as possible in safe languages to benefit from their extra language-enforced guarantees. These guarantees hold only for code that enforces them. For example, a C# or Java program linking to a C library must trust that the C code does not contain any memory-safety bugs because a single memory-safety error can invalidate the invariants in the safe part of the program.

Rewriting all of the existing code in a safe language is usually not economically feasible even in the cases where it is technically possible. Waiting until everything in a program is written in a safe language before being able to claim security properties arising from the safe language is therefore not a valid option. Instead, we would like to explore options for using unsafe code in safe ways. This problem requires exploring multiple levels of the stack, including the safe languages' abstract machines and the OS and hardware mechanisms used for isolation.

This discussion was driven by several high-level questions, covered in the following subsections.

3.5.1 What would you like to see in the abstract machines of new languages to facilitate interoperability?

This discussion was partly related to previous discussion on verification techniques. Linear memory and linear capabilities provide a clear transfer-of-ownership model that is easy to reason about. Once memory has been transferred to an untrusted domain, the trusted portion of a system does not have to enforce any guarantees for it. Transferring memory back is more complicated and requires enforcement mechanisms to prevent the untrusted code being able to access it.

Garbage collection poses some challenges for interoperability. Unsafe code must not be able to materialise pointers to garbage-collected memory and the garbage collector must be able to see and invalidate all pointers in untrusted memory. In the context of a model like that of WebAssembly, is it possible to provide garbage collector as a service and prove (and then rely on) properties of it?

Within the context of language abstractions there is a large open question: what properties should be statically or dynamically enforced and in which contexts? For example, a common compilation target (along the lines of Java or CLR bytecode) that has a strong type system can guarantee a lot of properties (at the expense of requiring that these properties hold for every source language) without the need for dynamic checks. Some dynamic checks can be offloaded to hardware, for example CHERI can dynamically enforce spatial safety even on uncooperative and untrusted machine code. Statically typed assembly languages have a long history and allow interoperating code to make stronger assumptions, should we be advocating for their use in all compilation chains, including those for unsafe languages?

A purely static approach to memory safety would require a very sophisticated type system, which might not even be feasible in the general case. A strong type system might be useful for optimization, where a language that statically guarantees certain properties can elide dynamic checks. For example, a language without linear types would need indirection and dynamic checks to guarantee that it enforced linearity at the boundary. There are still open questions, even assuming the existence of such a type system, that existing code could have dynamic checks inserted to be able to enforce useful properties at the interfaces.

WebAssembly, for example is typed (but its type system is simple) and therefore still needs dynamic checks. This provides some evidence for the amount of typing that compilers from C-like languages are willing to insert. Even simple properties, such as existential types, are probably difficult to drive to universal adoption. Implementing a garbage collector on top of WebAssembly is almost impossible because the type system does not differentiate between pointers and integers. This would become feasible with something like MSWasm, though some experiments implementing garbage collectors on top of CHERI suggest that most C/C++ code is not correct in the presence of copying garbage collection and so one would be restricted to non-moving collectors.

3.5.2 What guarantees in existing languages and abstract machines would you like to be able to protect when composing languages?

For safe composition, the source language needs some kind of many-worlds abstraction. Java has this at the abstract-machine level. The JNI defines an interface that allows Java code to call native code and for the native code to interact with the JVM, but in typical implementations there is no isolation enforced at the boundary. CHERI can be used to retrofit strong enforcement at this kind of boundary but a two-world abstraction doesn't provide any useful fault isolation in the unsafe world: any native code can compromise any other native code.

Memory safety was identified as the key building block for safe composition. Even in the absence of stronger guarantees, ensuring that a C module cannot access any memory that is not explicitly passed to it allows a safe language to make strong guarantees about the damage that a bug in the C code can do. Linearity, though not essential, was the top of the nice-to-have list, giving a simple mechanism for ensuring temporal non-interference between safe and unsafe components.

3.5.3 What features of existing languages would you like to be able to expose in other languages?

As before in this discussion, linearity was top of the list for many participants. Stepping back to think less specifically about source-language properties, there was one high-level guarantee that the participants agreed was critical, the Vegas Principle: What happens in an unsafe language, stays in the unsafe language. More specifically **no sequence of operations in one language may alter state shared between multiple languages in a way that is not possible in all of the sharing languages**.

Implementing secure interoperability is difficult if the abstraction is a *Foreign Function Interface* (FFI), because there is no associated notion of state ownership with a function-call abstraction. Ideally, we would have **Foreign Library Interface** (FLI) where one language could instantiate components in another language and then invoke functions within the scope of that instantiation. This is the model that the picoprocess abstraction, from the Bytecode Alliance, and Project Verona, from Microsoft Research, are attempting to adopt.

A shared-nothing design, such as that used by Erlang's to provide actors written in other languages, is the simplest to secure. All communication requires a copy (at least at the abstract-machine level) and so provides an explicit interception point. This also makes it plausible to compose memory-management policies. Some components can have garbage-collected private memory, others can have manual memory management, reference counting, or linear types. There is no need for a global garbage collector unless a language with automatic memory management can observe all memory.

The CHERI project has done a number of experiments on compartmentalization overheads. In many cases on existing systems, the overhead comes from defensive copies. Hardware acceleration for read-only sharing or ownership transfer would eliminate a lot of these.

3.5.4 What would language designers and implementers like hardware designers to provide?

The participants at this seminar spanned hardware and software and so this discussion provided an opportunity for software-facing researchers and practitioners to present a wishlist of features and for those on the hardware side to provide feedback on their feasibility.

As before, linear types were a popular request. There have been several proposals for linear capabilities on top of CHERI since around 2013. At the hardware level, these are not very difficult. The hardware would clear the tag if a linear capability were loaded, zero the source register on store, and require atomic exchange operations to load a usable linear capability. The problems typically arise in the operating system and compiler. Even in languages with linear types, compilers assume that they can duplicate register values, for example spilling a pointer of a linear type to the stack, using it, and then clobbering it in a subsequent instruction. Similarly, context-switch code, signal, and similar abstractions in the operating system assume that it's safe to load and store register values. Supporting hardware-enforced linear capabilities would require invasive changes to the entire software stack.

There have also been various proposals for restricting information flow in CHERI. The current proposal has a 2-bit information-flow policy, with a local/global split and a store-local permission such that local capabilities can be stored only through a capability with store-local permission. C has thread-local, stack, heap, and global objects with different lifetimes and so this policy is not sufficient for enforcing even this ordering (assuming C code that doesn't store stack pointers on the heap and so on). Capabilities for uninitialized memory that require every memory location to be stored through would help with some categories of vulnerability. These would allow a post-increment store on the address, but not loads until the address reached the end of the object.

There has also been a proposal for store-once capabilities. These contain a one-bit counter that is decremented on store. Attempting to store with the counter cleared would lose the tag bit. This scheme would prevent some exploit techniques. For example, a spilled return capability would be loadable multiple times but could not be stored to overwrite another spilled return capability with a valid capability.

WebAssembly was designed to be efficient to implement on current commodity hardware. The 32-bit memory model is partly accidental: it is easy to enforce on current hardware. Proposals to extend WebAssembly to support a 64-bit address space have suffered from a lack of efficient implementation techniques. Using multiple WebAssembly linear memories has suffered similarly.

In general, it is important to remember one fundamental rule for safe interoperability: **Isolation is easy, (safe) sharing is hard**. Full isolation can be achieved by complete physical partitioning of resources, but useful interoperability requires close communication, which typically implies data sharing (at least at the implementation level, even if the abstract machine describes copies). WebAssembly with WASI, for example, could be trivially implemented with native compilation and running the result in a process using Capsicum sandboxing. This would result in very fast WebAssembly execution but would require a full IPC and context switch for communication with the embedding environment, which would be too slow.

3.5.5 What do people see as the biggest open problems in language interoperability today?

From the formal side of the question, there are still a lot of open areas. If we want to define a formal model and proof of safe interoperability between two languages, is there a general understanding of what the proof structure should look like? Is there anything beyond multi-language semantics, for example capturing properties of the mechanisms used to enforce isolation?

On the practical side, there are very few examples of safe interoperability and most of them are research prototypes. For example, Robusta, Arabica, and CHERI-JNI all provided safe interoperability between Java and native code, but did not escape the lab. The examples that do exist are effectively different syntaxes on the same language. For example, Java and Kotlin both expose the JVM abstract machine, C# and F# the .NET abstract machine, and TypeScript compiles directly to JavaScript and so supports all of the target's semantics. Are there core abstractions that are both efficient to implement in hardware and provide useful guarantees for software engineering? Are sandboxing solutions part of this problem, providing coarse-grained isolation with explicit sharing?

The whole problem domain of secure compilation can be seen as an extreme case of safe interoperability, between a high-level language and a restricted subset of the target's functionality such that nothing in the output of a compiled program can violate the invariants of the source semantics.

4 Overview of Talks

4.1 Enforcement and compiler preservation of fine-grained constant-time policies

Gilles Barthe (MPI-SP – Bochum, DE)

License © Creative Commons BY 4.0 International license

© Gilles Barthe

Joint work of Gilles Barthe, Basavesh Ammanaghatta Shivakumar, Benjamin Gregoire, Vincent Laport, Swarn Priya

The constant-time (CT) policy is an information flow policy used by crypto libraries as a protection against cache-based side-channel attacks. At the core of the CT policy is a baseline leakage model, which assumes that only memory accesses and control flow are leaked. While this leakage model is adequate for analyzing many attacks from the literature: (a) it does not account for time-variable instructions, whose execution time depends on its operands; (b) it excludes real-world code, which uses a weaker leakage model and consequently achieves higher performance. We introduce a general class of fine-grained constant-time policies that supports both weaker and stronger leakage models and their combination. Then, we propose a two-step approach for enforcing fine-grained constant-time policies: first, prove that source programs are constant-time w.r.t. a fine-grained policy using relational Hoare logic, and then prove that compilation preserves constant-time w.r.t. a fine-grained policy. We implement the approach in the Jasmin framework for high-assurance cryptography. We use the framework to verify real-world cryptographic code that was out of the scope of previous approaches.

4.2 Formalizing Stack Safety as a Security Property

Roberto Blanco (MPI-SP – Bochum, DE)

License © Creative Commons BY 4.0 International license

© Roberto Blanco

Joint work of Roberto Blanco, Sean Anderson, Leonidas Lampropoulos, Benjamin Pierce, Andrew Tolmach

Main reference Sean Noble Anderson, Leonidas Lampropoulos, Roberto Blanco, Benjamin C. Pierce, Andrew Tolmach: “Security Properties for Stack Safety”, CoRR, Vol. abs/2105.00417, 2021.

URL <https://arxiv.org/abs/2105.00417>

What does “stack safety” mean, exactly? The phrase is associated with a variety of compiler, run-time, and hardware mechanisms for protecting stack memory, but these mechanisms typically lack precise specifications, relying instead on informal descriptions and examples of the bad behaviors that they prevent.

We propose a generic, formal characterization of stack safety based on concepts from language-based security: a combination of an integrity property (“the private state in each caller’s stack frame is held invariant by the callee”), and a confidentiality property (“the callee’s behavior is insensitive to the caller’s private state”), which can optionally be extended with a control flow property.

We use these properties to validate the stack-safety micro-policies proposed by Roessler and DeHon. Specifically, we check (with property-based random testing) that their “eager” micro-policy, which catches violations as early as possible, enforces a simple “stepwise” variant of our properties, and that (a repaired version of) their more performant “lazy” micro-policy enforces a slightly weaker and more extensional observational property. Meanwhile our testing successfully detects violations in several broken variants, including Roessler and DeHon’s original lazy policy.

4.3 Are Compiler Optimizations Doing it Wrong? An Investigation of Array Bounds Checking Elimination

Stefan Brunthaler (Universität der Bundeswehr – München, DE)

License © Creative Commons BY 4.0 International license
© Stefan Brunthaler

At the 2018 Secure Compilation meeting, I had several interesting discussions, which eventually lead to the realization that compiler optimizations usually have no stated threat model and are thus assuming overly benign operating conditions that do not withstand scrutiny at a closer look. In this talk, I present my preliminary analysis of a popular array bounds check elimination algorithm, ABCD.

4.4 Cross-Language Attacks

Nathan Burow (MIT Lincoln Laboratory – Lexington, US)

License © Creative Commons BY 4.0 International license
© Nathan Burow
Joint work of Nathan Burow, Hamed Okhravi, Samuel Mergendahl
Main reference Samuel Mergendahl, Nathan Burow, Hamed Okhravi: “Cross-Language Attacks”, Proceedings of the Network and Distributed System Security Symposium (NDSS’22), San Diego, CA, 2022

Memory corruption attacks against unsafe programming languages like C/C++ have been a major threat to computer systems for multiple decades. Various sanitizers and runtime exploit mitigation techniques have been shown to only provide partial protection at best. Recently developed “safe” programming languages such as Rust and Go hold the promise to change this paradigm by preventing memory corruption bugs using a strong type system and proper compile-time and runtime checks. Gradual deployment of these languages has been touted as a way of improving the security of existing applications before entire applications can be developed in safe languages. This is notable in popular applications such as Firefox and Tor. In this paper, we systematically analyze the security of multi-language applications. We show that because language safety checks in safe languages and exploit mitigation techniques applied to unsafe languages (e.g., Control-Flow Integrity) break different stages of an exploit to prevent control hijacking attacks, an attacker can carefully maneuver between the languages to mount a successful attack. In essence, we illustrate that the incompatible set of assumptions made in various languages enables attacks that are not possible in each language alone. We study different variants of these attacks and analyze Firefox to illustrate the feasibility and extent of this problem. Our findings show that gradual deployment of safe programming languages, if not done with extreme care, can indeed be detrimental to security.

4.5 Securing Interruptible Enclaved Execution on Small Microprocessors

Matteo Busi (University of Pisa, IT)

License  Creative Commons BY 4.0 International license
© Matteo Busi

Joint work of Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, Frank Piessens

Main reference Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, Frank Piessens: “Securing Interruptible Enclaved Execution on Small Microprocessors”, *ACM Trans. Program. Lang. Syst.*, Vol. 43(3), pp. 12:1–12:77, 2021.


URL <https://doi.org/10.1145/3470534>

Computer systems often provide hardware support for isolation mechanisms like privilege levels, virtual memory, or enclaved execution. Over the past years, several successful software-based side-channel attacks have been developed that break, or at least significantly weaken the isolation that these mechanisms offer. Extending a processor with new architectural or micro-architectural features brings a risk of introducing new software-based side-channel attacks.

In this talk we show how we extended a processor with new features without weakening the security of the isolation mechanisms that the processor offers. Our solution is heavily based on techniques from research on programming languages. More specifically, we propose to use the programming language concept of full abstraction as a general formal criterion for the security of a processor extension. We instantiate the proposed criterion to the concrete case of extending a microprocessor that supports enclaved execution with secure interruptibility. This is a very relevant instantiation as several recent papers have shown that interruptibility of enclaves leads to a variety of software-based side-channel attacks. We propose a design for interruptible enclaves, prove that it satisfies our security criterion and explain how such design drove the actual implementation of an enclave-enabled microprocessor.

4.6 Project Verona: An abstract machine allowing partial verification

David Chisnall (Microsoft Research – Cambridge, UK)

License  Creative Commons BY 4.0 International license
© David Chisnall

URL <https://github.com/Microsoft/Verona>

Project Verona is a project by MSR in collaboration with various academic partners to build a new secure programming language for large-scale infrastructure. Verona aims to eliminate any “unsafe” escape hatches so that the type safety and concurrency safety guarantees exist even in the presence of existing C/C++ libraries.

Verona has a “many worlds” abstract machine, providing isolation at the type-system level for units of concurrent execution and for data structures that can be transferred between units of execution. For pure Verona code, we aim to enforce all of these guarantees statically in the compiler, rejecting programs that would violate them. For programs using foreign libraries, we aim to use the type system to define where we must add dynamic checks.

Verona intends to expose instances of foreign libraries as regions, with an isolated memory space (containing the library’s heap, stack, globals, and so on) for each instance. This can be enforced with various techniques, such as processes, other MMU-based isolation, SFI, CHERI. The guarantees exposed from Verona to the foreign code are similarly strong: No concurrent access, no out-of-bounds memory accesses. Verona is currently an early-stage research project, this talk aims to provide a taste of the guarantees that we expect to be able to provide to people who will be able to use them as a building block for partially verified systems.

4.7 On information flow preserving refinement

Mads Dam (KTH Royal Institute of Technology – Stockholm, SE)

License © Creative Commons BY 4.0 International license
© Mads Dam

Joint work of Mads Dam, Christoph Baumann, Hamed nemati, Roberto Guanciale

Main reference Christoph Baumann, Mads Dam, Roberto Guanciale, Hamed Nemati: “On Compositional Information Flow Aware Refinement”, in Proc. of the 34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021, pp. 1–16, IEEE, 2021.

URL <https://doi.org/10.1109/CSF51468.2021.00010>

Information flow security and refinement have had a troublesome relationship since many years. Refinement injects implementation decisions that in general will cause information content to increase and, as a consequence, can cause information flow properties to be violated. How to address this in a way that supports the many use cases of refinement (changes in data representation, reduction of nondeterminism/underspecification, addition of new observation variables, for instance to reflect low-level features such as caches) has remained open for many years. Building on initial work by Morgan we propose a new approach based on ignorance preservation: A refinement step should be viewed as information flow preserving, if it does not cause observers ignorance to be reduced, for instance by revealing some secret bit of information. In the talk we present the basic epistemic set-up, give some examples, and discuss different proof methods and complications related, in particular, to compositionality.

4.8 Formalizing ISA security guarantees in the form of universal contracts

Dominique Devriese (KU Leuven, BE)

License © Creative Commons BY 4.0 International license
© Dominique Devriese

Joint work of Dominique Devriese, Sander Huyghebaert, Steven Keuchel

Where ISA specifications used to be defined in long prose documents, we have recently seen progress on formal and executable ISA specifications. However, for now, formal specifications provide only a functional specification of the ISA, without specifying the ISA’s security guarantees. In this paper, we present a novel, general approach to specify an ISA’s security guarantee in a way that (1) can be semi-automatically validated against the ISA semantics, producing a mechanically verifiable proof, (2) supports informal and formal reasoning about security-critical software in the presence of adversarial code. Our approach is based on the use of universal contracts: software contracts that express bounds on the authority of arbitrary untrusted code on the ISA. We semi-automatically verify these contracts against existing ISA semantics implemented in Sail using our Katamaran tool: a verified, semi-automatic separation logic verifier for Sail. For now, in this paper, we will illustrate our approach for MinimalCaps: a simplified custom-built capability machine ISA. However, we believe our approach has the potential to redefine the formalization of ISA security guarantees and we will sketch our vision and plans.

4.9 Proof techniques for secure compilation with memory sharing

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

License © Creative Commons BY 4.0 International license
© Akram El-Korashy

Joint work of Akram El-Korashy, Adrien Durier, Roberto Blanco, Deepak Garg, Catalin Hritcu, Jeremy Thibault
Main reference Akram El-Korashy, Roberto Blanco, Jérémy Thibault, Adrien Durier, Deepak Garg, Catalin Hritcu:
“SecurePtrs: Proving Secure Compilation with Data-Flow Back-Translation and Turn-Taking
Simulation”, CoRR, Vol. abs/2110.01439, 2021.

URL <https://arxiv.org/abs/2110.01439>

In two recent pieces of work, we studied techniques for proving two theorems about secure compilation of partial programs (namely, a compiler full abstraction and a preservation of robust safety theorem). Secure compilation of partial programs aims to defend against adversarial contexts (e.g. untrusted libraries). We focus on settings in which the compiled partial program is allowed to share – at run-time – parts of its memory with the context by pointer passing (and the context is also allowed the same).

Proving secure compilation of partial programs typically requires back-translating a target attack against the compiled program to an attack against the source program. To prove this back-translation step, we propose a new technique called data-flow back-translation that is simple, handles unstructured control flow and memory sharing well, and we have proved it correct in Coq.

Our proof techniques work without relying on any assumption about the behavior of the context, but they do rely on target-language support that enforces spatial memory safety. I will present the proof techniques and explain how they allow reusing whole-program compiler correctness proofs. Such reuse is novel, especially for settings with memory sharing, and it is practically desirable in order to avoid redoing laborious proofs should a compiler correctness theorem already exist.

4.10 Preserving Memory Safety from C to MSWasm

Anitha Gollamudi (Yale University – New Haven, US)

License © Creative Commons BY 4.0 International license
© Anitha Gollamudi

Joint work of Anitha Gollamudi, Marco Vassena, Marco Patrignani, Deian Stefan, Craig Disselkoen, Alexandra Michael, Aidan Denlinger, Bryan Parno, Jay Bosamiya, Conrad Watt

WebAssembly (Wasm) has gained traction as the new portable compilation target language for deploying on the web applications written in high-level source languages like C, C++, and Rust. Memory safety is key to the isolation mechanism of the sandboxed execution environment: well-typed programs cannot corrupt the memory outside the sandbox (e.g., the Javascript virtual machine). Unfortunately, Wasm is still insecure: buffer overflows and use-after-free can still corrupt the memory of a program within the sandbox, opening the door to attacks like cross-site scripting and remote code execution.

In this talk we present Memory-Safe WebAssembly (MSWasm), an extension of Wasm with built-in spatial memory safety. That is, any well-typed program in MSWasm is proven to attain spatial memory safety robustly, i.e., even in the presence of arbitrary code the program links against. Additionally, we show that MSWasm can be used as a compilation target for C programs.

Our MSWasm development is built with solid formal foundations: we provide a formal model of MSWasm which we use to prove robust spatial memory safety; we formalise our compiler from (a subset of) C to MSWasm and prove that the compiler is not just correct, but it preserves memory safety of C programs into their MSWasm counterparts; and provide an implementation for C to MSWasm as well as benchmark its efficiency.

4.11 Contract-aware secure compilation: a foundation for side-channel resistant compilers – Challenges and open questions

Marco Guarnieri (*IMDEA Software – Madrid, ES*)

License  Creative Commons BY 4.0 International license
© Marco Guarnieri

In the talk, I discussed how compilers can help in building systems that are secure against side-channel and microarchitectural attacks. For this, I presented an overview of hardware-software contracts: an abstraction that captures a processor’s security guarantees in a simple, mechanism-independent manner by specifying which program executions a microarchitectural attacker can distinguish. Next, I introduced the idea of contract-aware secure compilation (CASCO). Contract-aware compilers leverage the guarantees expressed in a given contract to generate code that is free from microarchitectural leaks. This enables decoupling program-level security (e.g., ensuring that a password is not leaked under the program semantics), which is the programmer’s responsibility, from microarchitectural security (e.g., ensuring that a password is not leaked due to microarchitectural side-effects), which is automatically enforced by compilers. I concluded by discussing challenges and open questions that needs to be solved for building CASCO compilers.

4.12 Formally verifying a secure compilation chain for unsafe C components

Catalin Hritcu (*MPI-SP – Bochum, DE*)

License  Creative Commons BY 4.0 International license
© Catalin Hritcu

Joint work of Catalin Hritcu, Arthur Azevedo de Amorim, Roberto Blanco, Akram El-Korashy, Deepak Garg, Marco Patrignani, Jeremy Thibault, Carmine Abate, Ștefan Ciobăcă, Adrien Durier, Boris Eng, Ana Nora Evans, Guglielmo Fachini, Théo Laurent, Benjamin C. Pierce, Marco Stronati, Éric Tanter, 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>

Undefined behavior is widespread in the C language and leads to devastating security vulnerabilities. We study how compartmentalization can mitigate this problem by restricting the scope of undefined behavior both (1) spatially to just the components that encounter undefined behavior and (2) temporally by still providing protection to each component up to the point in time when it encounters undefined behavior and becomes compromised. In this talk, we report on a project that has been ongoing for over 5 years on building a formally verified secure compilation chain for unsafe C components based on a variant of the CompCert compiler and various low-level enforcement mechanisms.

We discuss how far did we get and what were the main challenges we had to overcome: from defining formally what it means for a compilation chain to be secure in this setting, to devising more scalable proof techniques that also allow sharing memory dynamically by passing pointers between components, from mechanizing our proofs in the Coq proof assistant, to supporting multiple enforcement mechanisms such as SFI and a programmable tagged architecture. We conclude with future work specific to our project as well as more general open challenges.

4.13 Conditional Contextual Refinement

Chung-Kil Hur (Seoul National University – Seoul, KR)

License  Creative Commons BY 4.0 International license
© Chung-Kil Hur

Joint work of Chung-Kil Hur, Youngju Song, Minki Cho, Dongjae Lee

Contextual refinement (CR) is one of the standard notions of specifying open programs. CR has two main advantages: (i) (horizontal and vertical) compositionality that allows us to decompose a large contextual refinement into many smaller ones enabling modular and incremental verification, and (ii) no restriction on programming features thereby allowing, e.g., mutual recursive, pointer-value passing, and higher-order functions. However, CR has a downside that it cannot impose conditions on the context since it quantifies over all contexts, which indeed plays a key role in support of full compositionality and programming features.

In this work, we address the problem of finding a notion of refinement that satisfies all three requirements: support of full compositionality, full (sequential) programming features, and rich conditions on the context. As a solution, we propose a new theory of refinement, called CCR (Conditional Contextual Refinement), and develop a verification framework based on it, which allows us to modularly and incrementally verify a concrete module against an abstract module under separation-logic-style pre and post conditions about external modules. It is fully formalized in Coq and provides a proof mode that combines (i) simulation reasoning about preservation of side effects such as IO events and termination and (ii) propositional reasoning about pre and post conditions. Also, the verification results are combined with CompCert, so that we formally establish behavioral refinement from top-level abstract programs, all the way down to their assembly code.

4.14 CompCertO: Compiling Certified Open C Components

Jérémie Koenig (Yale University – New Haven, US)

License  Creative Commons BY 4.0 International license
© Jérémie Koenig

Joint work of Jérémie Koenig, Zhong Shao

Main reference Jérémie Koenig, Zhong Shao: “CompCertO: compiling certified open C components”, in Proc. of the PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021, pp. 1095–1109, ACM, 2021.

URL <https://doi.org/10.1145/3453483.3454097>

Since the introduction of CompCert, researchers have been refining its language semantics and correctness theorem, and used them as components in software verification efforts. Meanwhile, artifacts ranging from CPU designs to network protocols have been successfully verified, and there is interest in making them interoperable to tackle end-to-end verification at

an even larger scale. Recent work shows that a synthesis of game semantics, refinement-based methods, and abstraction layers has the potential to serve as a common theory of certified components. Integrating certified compilers to such a theory is a critical goal. However, none of the existing variants of CompCert meets the requirements we have identified for this task. CompCertO extends the correctness theorem of CompCert to characterize compiled program components directly in terms of their interaction with each other. Through a careful and compositional treatment of calling conventions, this is achieved with minimal effort.

4.15 Changing Compilation without Changing the Compiler

Per Larsen (Immunant – Irvine, US)

License © Creative Commons BY 4.0 International license
© Per Larsen

Joint work of Per Larsen, Andrei Homescu, Stephen Crane

Lots of security research requires changing how compilation is done. For prototyping purposes, this is usually done by downloading the source code of the compiler, etc. If the underlying techniques were ever to be put into practice, we run into the problem that different folks use different compilers. This talk covers a few projects that ran into this challenge and what one can do to avoid the need to customize the compiler. Specifically, I will cover a code randomization project that wraps the linker to rewrite the output of the compiler and a compartmentalization project that rewrites C/C++ headers to avoid modifying the compiler.

4.16 WebAssembly as an intermediate language for safe interoperability

Zoe Paraskevopoulou (Northeastern University – Boston, US)

License © Creative Commons BY 4.0 International license
© Zoe Paraskevopoulou

Joint work of Zoe Paraskevopoulou, Amal Ahmed, Michael Fitzgibbons

In this talk I discussed ongoing work on WebAssembly that focuses on enhancing WebAssembly with capabilities (static and dynamic) in order to facilitate interoperability between languages with different features.

4.17 Compositional Secure Compilation against Spectre

Marco Patrignani (CISPA – Saarbrücken, DE)

License © Creative Commons BY 4.0 International license
© Marco Patrignani

Joint work of Marco Patrignani, Marco Guarnieri, Xaver Fabian, Matthis Kruse

Main reference Marco Guarnieri, Marco Patrignani: “Exorcising Spectres with Secure Compilers”, CoRR, Vol. abs/1910.08607, 2019.


URL <http://arxiv.org/abs/1910.08607>

I reported on the CCS’21 paper on the secure compilation against spectre v1 and then talk about how we want to scale these results to v2, v4, v5 and their compositions (e.g., proving that a compiler is secure against v1+v4 simultaneously). Doing this requires reasoning compositionally about robust compilation and what properties we want to preserve, which

is an interesting extension of the robust compilation line of work. I also spoke about this compositionality issue in more general terms, trying to generalise these results beyond preservation of Spectre security to the preservation of other security properties.

4.18 Automatic inference of effective compartmentalization policies

Mathias Payer (EPFL – Lausanne, CH)

License  Creative Commons BY 4.0 International license
© Mathias Payer

Severe vulnerabilities are continuously discovered in low level code. Those vulnerabilities threaten the confidentiality and integrity of our systems. Compartmentalization enforces isolation between components and allows breaking up large complex systems into small trust compartments that contain any faults.

While different efficient compartmentalization mechanisms exist, developing effective policies is challenging and generally remains a manual process. In a study on the Linux kernel, we evaluate the feasibility of simple directory-based policies and develop a framework for reasoning about memory accesses during program execution [1].

Shifting towards generating policies, we propose two approaches that leverage the language environment to implement effective compartmentalization. First, HAKCs [2] targets the Linux kernel and allows developers to specify ownership for data along with passing said data between strictly enforced compartments. This explicit ownership model enables efficient checks but revoking privileges for aliased pointers remains challenging. In a prototype targeting the IPv6 module, we demonstrate how such compartmentalization is feasible. Second, in programming environments that heavily rely on third-party libraries, trusting said libraries remains challenging and this trust may be broken by arbitrary updates that are outside of the control of the software developer. With Enclosures [3], we enable flexible closures that bind calls across library boundaries to dynamically created compartments with access to limited system calls. The address space of the process is shared, in part, with the compartment so that data exchange effectively functions.

Finally, we discuss extensions to existing architectures that enable efficient sharing between compartments. Our RISC V prototype leverages a metadata table that defines compartments inside an address space (similar, at a high level, to flexible segments with an inter-segment switching policy), introducing unprivileged instructions that securely switch between compartments.

The discussion centered around effective implementations, creation of strict policies, and limitations of different hardware extensions. Compartmentalization is an active research area that profits from advances in secure compilation in several ways, namely through policy generation, inference of compartment boundaries, and integration as well as activation of the different hardware extensions. Many challenging open questions remain in this active research area.

References

- 1 uSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P. Kemerlis, Mathias Payer, Adam Bates, Andre DeHon, Jonathan M. Smith, and Nathan Dautenhahn. In RAID'21: Recent Advances in Intrusion Detection, 2021

- 2 Preventing Kernel Hacks with HAKCs. Derrick McKee, Yianni Giannaris, Carolina Ortega, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. In NDSS'22: Network and Distributed System Security Symposium, 2022
- 3 Enclosure: language-based restriction of untrusted libraries. Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. In ASPLOS'21: International Conference on Architectural Support for Programming Languages and Operating Systems, 2021

4.19 Hardware-Software Contracts and Secure Programming

Jan Reineke (Universität des Saarlandes – Saarbrücken, DE)

License © Creative Commons BY 4.0 International license
© Jan Reineke

Joint work of Jan Reineke, Marco Guarnieri, Boris Köpf, Pepe Vila

Main reference Marco Guarnieri, Boris Köpf, Jan Reineke, Pepe Vila: “Hardware-Software Contracts for Secure Speculation”, in Proc. of the 2021 IEEE Symposium on Security and Privacy (SP), pp. 1868–1883, 2021.

URL <https://doi.org/10.1109/SP40001.2021.00036>

Cache attacks and more recently transient-execution attacks demonstrate that microarchitectural components may leak information in unintended and surprising ways. I will discuss recent work on formally capturing microarchitectural leakage using hardware-software contracts with the goal of enabling secure programming.

4.20 Hardware-assisted testing in production

Kostya Serebriany (Google – Mountain View, US)


License © Creative Commons BY 4.0 International license
© Kostya Serebriany

Every software vendor is trying to “shift left”, i.e. to move bug detection to earlier stages of software development. This is an important goal, which we are unlikely to ever achieve 100%, and thus we need to keep finding bugs when the software is already released. In this talk we will discuss three testing mechanisms that use special hardware features to enable sampled bug detection with near-zero overhead in production:

- GWP-ASan, detects heap corruption with hardware page protection.
- Per-allocation sampling with Arm Memory Tagging Extension.
- GWP-TSan, detects data races using hardware watchpoints.

4.21 Morello status and verification

Peter Sewell (University of Cambridge, UK)

License  Creative Commons BY 4.0 International license
 Peter Sewell

Main reference Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, Peter Sewell: “Verified security for the Morello capability-enhanced prototype Arm architecture”, 2021.

URL <https://doi.org/10.48456/tr-959>

I gave an update on the state of Morello, the Arm prototype architecture and processor incorporating CHERI hardware capability support, and on our work to verify fundamental properties of the full 62k LoS architecture specification.

4.22 A Wishlist for the Next Generation of Trusted Execution Environments



Shweta Shinde (ETH Zürich, CH)

License  Creative Commons BY 4.0 International license
 Shweta Shinde

I highlighted the ongoing initiatives and research directions for building the next generation of trusted execution environments with the primary goal of supporting verified secure software.

4.23 Swivel: Hardening WebAssembly against Spectre

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

License  Creative Commons BY 4.0 International license
 Deian Stefan

Joint work of Deian Stefan, Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen

Main reference Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, Deian Stefan: “Swivel: Hardening WebAssembly against Spectre”, in Proc. of the 30th USENIX Security Symposium (USENIX Security 21), pp. 1433–1450, USENIX Association, 2021.

URL <https://www.usenix.org/conference/usenixsecurity21/presentation/narayan>

We describe Swivel, a new compiler framework for hardening WebAssembly (Wasm) against Spectre attacks. Outside the browser, Wasm has become a popular lightweight, in-process sandbox and is, for example, used in production to isolate different clients on edge clouds and function-as-a-service platforms. Unfortunately, Spectre attacks can bypass Wasm’s isolation guarantees. Swivel hardens Wasm against this class of attacks by ensuring that potentially malicious code can neither use Spectre attacks to break out of the Wasm sandbox nor coerce victim code—another Wasm client or the embedding process—to leak secret data. We describe two Swivel designs, a software-only approach that can be used on existing CPUs, and a hardware-assisted approach that uses extension available in Intel 11th generation CPUs. For both, we evaluate a randomized approach that mitigates Spectre and a deterministic approach that eliminates Spectre altogether. Our randomized implementations impose under 10.3% overhead on the Wasm-compatible subset of SPEC 2006, while our deterministic implementations impose overheads between 3.3% and 240.2%. Though high on some benchmarks, Swivel’s overhead is still between 9x and 36.3x smaller than existing defenses that rely on pipeline fences.

4.24 Compiler-based Side Channel Detection and Mitigation

Gang Tan (Pennsylvania State University – University Park, US)

- License** © Creative Commons BY 4.0 International license
© Gang Tan
- Main reference** Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, Mahmut Kandemir: “CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation”, in Proc. of the 2019 IEEE Symposium on Security and Privacy (SP), pp. 505–521, 2019.
URL <https://doi.org/10.1109/SP.2019.00022>
- Main reference** Robert Brotzman, Danfeng Zhang, Mahmut Taylan Kandemir, Gang Tan: “SpecSafe: detecting cache side channels in a speculative world”, Proc. ACM Program. Lang., Vol. 5(OOPSLA), pp. 1–28, 2021.
URL <https://doi.org/10.1145/3485506>

We describe two systems (CaSym and SpecSafe), which use symbolic execution to detect and mitigate cache-based side channels in software or verify their absence. We will also discuss what components are needed to achieve secure compilation in the presence of side channels caused by conventional or speculative execution.

4.25 Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code

Thomas Van Strydonck (KU Leuven, BE)

- License** © Creative Commons BY 4.0 International license
© Thomas Van Strydonck
- Joint work of** Thomas Van Strydonck, Lars Birkedal, Dominique Devriese, Armaël Guéneau, Aïna Linn Georges, Amin Timany, Alix Trieu
URL <https://github.com/logsem/cerise>

A capability machine is a type of CPU allowing fine-grained privilege separation using capabilities, machine words that represent certain kinds of authority. We present Cerise, a mathematical model and accompanying proof methods that can be used for formal verification of functional correctness of programs running on a capability machine, even when they invoke and are invoked by unknown (and possibly malicious) code. Our work has been entirely mechanized in the Coq proof assistant using the Iris program logic framework. The methodology we present underlies recent work of the authors on formal reasoning about capability machines, but was left somewhat implicit in those publications. This presentation exposes in further details a pedagogical introduction to the methodology, in a simple setting (no exotic capabilities), and starting from minimal examples.

4.26 Software Supply Chains: Challenges and Opportunities


Nikos Vasilakis (MIT – Cambridge, US)

- License** © Creative Commons BY 4.0 International license
© Nikos Vasilakis

To lower the time and cost of engineering software, developers today use software supply chains of unprecedented scale: It is not uncommon for a modern application to use hundreds or even thousands of third-party dependencies developed by many developers with varying needs, skill levels, care, and intentions. In this talk, I will outline some of the security challenges associated with third-party dependencies, and show how key characteristics of these dependencies enable program-transformation techniques to overcome these challenges while keeping developer burden manageable.

4.27 How we design hardware and what is costs?


Ingrid Verbauwhede (KU Leuven, BE)

License  Creative Commons BY 4.0 International license
© Ingrid Verbauwhede

I gave basics on what are the hardware design constraints: how we measure performance, throughput, latency.

4.28 Verifying Speculation Security of Processor Implementations

Drew Zagieboylo (Cornell University – Ithaca, US)

License  Creative Commons BY 4.0 International license
© Drew Zagieboylo
Joint work of Drew Zagieboylo, Edward Suh, Andrew Myers

We discuss existing tools for verifying security properties in RTL designs and their applicability to speculation-aware contracts. In particular, we highlight the difficulty of verifying speculation security since it is often dependent upon verifying functional correctness. Existing tools either require significant manual input which cannot easily be re-purposed across designs, or they involve assumptions which are difficult to trust or reason about in complex designs.

We propose integrating domain knowledge of speculative processor design directly into a higher-level hardware description language to simplify correctness and speculative-security reasoning. We hope to limit difficult RTL verification tasks to modularized components that abstract common microarchitectural optimizations such as bypassing, speculation, and instruction re-ordering.

Participants

- Roberto Blanco
MPI-SP – Bochum, DE
- Stefan Brunthaler
Universität der Bundeswehr –
München, DE
- Matteo Busi
University of Pisa, IT
- Dominique Devriese
KU Leuven, BE
- Akram El-Korashy
MPI-SWS – Saarbrücken, DE
- Deepak Garg
MPI-SWS – Saarbrücken, DE
- Anitha Gollamudi
Yale University – New Haven, US
- Marco Guarnieri
IMDEA Software – Madrid, ES
- Catalin Hritcu
MPI-SP – Bochum, DE
- Marco Patrignani
CISPA – Saarbrücken, DE
- Jan Reineke
Universität des Saarlandes –
Saarbrücken, DE
- Shweta Shinde
ETH Zürich, CH
- Jeremy Thibault
MPI-SP – Bochum, DE
- Thomas Van Strydonck
KU Leuven, BE
- Ingrid Verbauwhede
KU Leuven, BE



Remote Participants

- Amal Ahmed
Northeastern University –
Boston, US
- Arthur Azevedo de Amorim
Boston University, US
- Gilles Barthe
MPI-SP – Bochum, DE
- Joseph Bialek
Microsoft – Redmond, US
- Sandrine Blazy
University & IRISA –
Rennes, FR
- Nathan Burow
MIT Lincoln Laboratory –
Lexington, US
- David Chisnall
Microsoft Research –
Cambridge, GB
- Mads Dam
KTH Royal Institute of
Technology – Stockholm, SE
- Ergys Dona
EPFL Lausanne, CH
- Cédric Fournet
Microsoft Research –
Cambridge, GB
- Tal Garfinkel
Corepoint Systems –
Penn Valley, US
- Chung-Kil Hur
Seoul National University, KR
- Jérémie Koenig
Yale University – New Haven, US
- Per Larsen
Immuntant – Irvine, US
- Amit Levy
Princeton University, US
- Toby Murray
The University of Melbourne, AU
- Andrew Myers
Cornell University – Ithaca, US
- Santosh Nagarakatte
Rutgers University –
Piscataway, US
- Elisabeth Oswald
Alpen-Adria-Universität
Klagenfurt, AT
- Zoe Paraskevopoulou
Northeastern University –
Boston, US
- Mathias Payer
EPFL – Lausanne, CH

■ Andreas Rossberg
Dfinity – Zürich, CH

■ Kostya Serebryany
Google – Mountain View, US

■ Peter Sewell
University of Cambridge, GB

■ Zhong Shao
Yale University – New Haven, US

■ Deian Stefan
University of California –
San Diego, US

■ Gang Tan
Pennsylvania State University –
University Park, US

■ Nikos Vasilakis
MIT – Cambridge, US

■ Marco Vassena
CISPA – Saarbrücken, DE

■ Drew Zagieboylo
Cornell University – Ithaca, US

