Efficient Formally Secure Compilation

Abstract Severe low-level vulnerabilities abound in today’s computer systems, allowing cyber-attackers to remotely gain full control. This happens in big part because our programming languages, compilers, and architectures were designed in an era of scarce hardware resources and too often trade off security for efficiency. The semantics of mainstream low-level languages like C is inherently insecure, and even for safer languages, establishing security with respect to a high-level semantics does not guarantee the absence of low-level attacks. Secure compilation using the coarse-grained protection mechanisms provided by mainstream hardware architectures would be too inefficient for most practical scenarios.

SECOMP is an ERC-funded project aimed at leveraging emerging hardware capabilities for fine-grained protection to build the first efficient secure compilation chains for realistic low-level programming languages (the C language, and Low* [110] a safe subset of C embedded in F* [119] for verification). These compilation chains will provide a more secure semantics for source programs and will ensure that high-level abstractions cannot be violated even when interacting with untrusted low-level code. To achieve this level of security without sacrificing efficiency, our secure compilation chains target a tagged architecture [22, 54], which associates a metadata tag to each word and efficiently propagates and checks tags according to a software-defined micro-policy. We use property-based testing and formal verification to provide high confidence that our compilers are indeed secure. Formally, we construct machine-checked proofs in Coq of various new security criteria, which are defined as the preservation of property classes even against an adversarial context. These strong criteria complement compiler correctness and ensure that no machine-code attacker can do more harm to securely compiled components than a source-level attacker already could with respect to a secure semantics for the source language.

Context: Low-level attacks on computer systems are a severe security problem. Today’s computer systems are distressingly insecure. This affects the foundation upon which today’s information society is built and makes everyone potentially vulnerable. Visiting a website, opening an email, or serving a client request is often enough to cause a computer to be compromised by a cyber-attack giving remote attackers full control. This often results in the disclosure or destruction of information and the use of the machine in further cyber-attacks. Hundreds of thousands of compromised computers are hoarded into “botnets” that are used to send spam, mount distributed denial of service attacks, or mine cryptocurrency. Given their cyber-attack power, previously unknown (“0-day”) exploitable low-level vulnerabilities in widely-used software are often sold to intelligence agencies or botnet “controllers” for tens to hundreds of thousands of dollars.

The causes for this dissatisfying state of affairs are complex, but at this point mostly historical: our programming languages, compilers, and architectures were designed in an era of scarce hardware resources and far too often trade off security for efficiency. Today’s mainstream low-level languages, C and C++, give up even on the most basic safety checks for the sake of efficiency, which leaves programmers bearing all the burden for security: the smallest mistake in widely-deployed C and C++ code can cause security vulnerabilities with disastrous consequences [58]. Over the last 12 years, around 70 percent of all patched security bugs at Microsoft were memory safety issues [96, 42]. The C and C++ languages do not guarantee memory safety because currently deployed hardware provides no good support for it and software checks would incur 70-80% overhead on average [100, 99]. Instead, much weaker low-overhead mitigation techniques are deployed and routinely circumvented by practical attacks [120, 60, 45]. Unfortunately, just ensuring memory safety would in fact not be enough to make C and C++ safe, as the standards and compilers for these languages call out a much larger number of undefined behaviors [68, 83], for which compilers produce code that behaves arbitrarily, often leading to security vulnerabilities, including invalid unchecked type casts [57, 67], data races, and sometimes even integer overflows.

Safer languages such as Java, C#, ML, Haskell, or Rust provide type and memory safety by default as well as many useful abstractions for writing more secure code (e.g., modules, interfaces, parametric polymorphism, etc). Unfortunately though, these languages are still not immune to low-level attacks. All the safety guarantees of these languages are lost when interacting with low-level code, for instance when using low-level libraries. This interaction is useful but dangerous because the low-level code can be malicious or compromised (e.g., by a buffer overflow). Currently, not only is the low-level code trusted to be safe, but also to preserve all the complex abstractions and internal invariants of the high-level language semantics, compiler, and runtime system. So even if some critical code is secure with respect to the semantics of a high-level language, any low-level code with which it interacts can break its security.

Verification languages such as Coq and F* [119] provide additional abstractions, such as dependent types, logical pre- and postconditions, and tracking of side effects, e.g., distinguishing pure from stateful computations. Such abstractions are crucial for making the verification effort more tractable in practice, but they also make the final verification result only valid in these very abstract languages. In order for a Coq or F* program to be executed it is first compiled all the way down to machine code. Even if such compilation is correct [91, 84], this is usually not enough to ensure the security of the verified code, since usually not all the code can be written and verified in the abstract verification language.

For a concrete example, consider the mTLS* implementation of the TLS standard, the most widely-used security protocol framework on the Internet. mTLS* is being written and formally verified in Low*, a safe subset of C embedded in F* [110, 37, 128]. mTLS* includes tens of thousands lines of Low* code, and even when all this code will be formally verified, it will
be just a tiny library linked from large unverified applications such as web browsers, web servers, and operating systems, which have millions of lines of C, C++, and assembly code. Not only are these applications not verified and can thus break the verified security properties of the Low* code, but these applications are not even memory safe, and any error can allow remote attackers to take complete control, disclose the memory of the process stealing the TLS private keys, etc. A correct compilation chain is not enough in this case, since (1) a correct compilation chain [91] for an insecure language like C still produces insecure code and leaves the burden of avoiding undefined behaviors to the programmer, and (2) a correct compilation chain does not protect the interaction between high-level and low-level code and does not enforce the abstractions of each language against faulty or malicious code written in the lower-level languages. In order for miTLS* to be secure in practice, we don’t need only correct compilation, but also secure compilation.

Secure compilation We call a compilation chain secure when it prevents or soundly mitigates low-level attacks on compiled code and it allows sound reasoning about security in the source language. Secure compilation (1) gives all programs in the source language a secure semantics (e.g., ensuring memory safety, component isolation, defined behavior, etc.) and (2) protects compiled programs from their interaction with unsafe low-level code.

SECOMP grand challenge: build the first efficient formally secure compilers for realistic programming languages. For this we will devise secure compiler chains from Low* [110] to C, and from C to RISC-V [18] assembly. These compiler chains will provide security for programs written in a combination of Low*, C, and assembly, by providing more secure semantics to C programs and by protecting the abstractions of each language against attacks from the lower-level ones. To achieve this level of security without sacrificing efficiency, our compiler chains will target our micro-policies tagged architecture [22]. For measuring and optimizing efficiency we will use standard benchmark suites [69] and realistic source programs, with miTLS* as the main end-to-end case study. In order to ensure high confidence in the security of our compiler chains, we will thoroughly test them using property-based testing [104] and then formally verify their security using Coq.

Further Studying Secure Compilation Criteria As discussed in Form 3, Fiche 2, so far most the work on this project has been focused on devising secure compilation criteria based on preserving classes of properties against adversarial contexts [81, 8] and extending these criteria to unsafe languages [7, 72]. Several interesting open problems remain in this space, including extending our component model [7] with dynamic component creation. This would make crucial use of our current dynamic compromise model, since components would no longer be statically known, and thus static compromise would not apply at all. This extension could allow us to move from our current “code-based” compartmentalization model to a “data-based” one [66], e.g., one compartment per incoming network connection or per browser tab.

Formally Secure Compartmentalization for C We plan to base our compartmentalizing compilation chain on the CompCert C compiler. Scaling up to the whole of C will certainly entail challenges such as defining a variant of C with components and efficiently enforcing compartmentalization all the way down using micro-policies. To achieve this, we will build on the solid foundation of our recent work [7]: the formal security criterion, the scalable proof technique, and the proof-of-concept secure compilation chain. As a first step, we plan to extend our prototype to allow sharing memory between components. Since we already allow arbitrary reads at the lowest level, it seems appealing to also allow external reads from some of the components’ memory in the source. The simplest would be to allow certain static buffers to be shared with all other components, or only with some if we also extend the interfaces. More ambitious would be to allow pointers to dynamically allocated memory to be passed to other components, as a form of read capabilities. This would make pointers appear in traces and we will need to take into account that these pointers will vary at the different levels in our compilation chain. Moreover, each component produced by the back-translation would need to record all the read capabilities it receives for later use. Finally, to safely allow write capabilities we could combine compartmentalization with memory safety. This will give us a fine-grained object-capability model [123] on a fully generic tagged architecture and will enable compartmentalized applications that are much more granular and secure than using currently-deployed isolation techniques [113, 66, 126].

Dynamic Privilege Notions Our prototype compilation chain [7] uses a very simple notion of interface to statically restrict the privileges of components. This could, however, be extended with dynamic notions of privilege such as capabilities and history-based access control [1]. In one of its simplest forms, allowing pointers to be passed between components and then used to write data, as discussed above, would already constitute a dynamic notion of privilege, that is not captured by the static interfaces, but nevertheless needs to be enforced, in this case using some form of memory safety.

Memory Safety for C We also plan to enforce memory safety for C and its interactions with untrusted RISC-V assembly. This will protect buggy programs from malformed inputs that would normally trigger undefined behavior. Enforcing memory safety requires changes to the C compiler and a sophisticated micro-policy, which extends our simple heap memory safety policy [23, 54, 19] to additionally deal with unboxed structs, stack allocation [115], byte addressing, unaligned memory accesses, custom allocators, etc. We plan to build an extension of CompCert [91] that is memory safe. To verify security we will target both properties describing the absence of spatial (e.g., buffer overflows) and temporal (e.g., use after free, double free) memory safety violations [100] and our higher-level reasoning principles enabled by memory safety [24].

Verifying Compartmentalized Applications. It would also be interesting to build verification tools based on the source reasoning principles provided by our secure compilation criteria [7] and to use these tools to analyze the security of practical compartmentalized applications. Effective verification, however, requires good ways for reasoning about the exponential number of dynamic compromise scenarios. One idea is to do our source reasoning with respect to a variant of our partial semantics, which would use nondeterminism to capture the compromise of components and their possible successive
actions. Correctly designing such a partial semantics for a complex language is, however, challenging. Fortunately, our secure compilation criteria provide a more basic, low-TCB definition against which to validate any fancier reasoning tools, like partial semantics [78], program logics [79], logical relations [52], etc.

**Secure compilation of Low* to C using Components, Contracts, and Sealing** We also plan to devise a secure compilation chain from Low* to C. Low* programs are verified with respect to Hoare-style pre- and post-conditions to achieve correctness and use the F* module system (i.e., data abstraction and parametricity) to achieve confidentiality of secret data, even against certain side-channels. These high-level abstractions will have to be protected at the C level, and while compartmentalization will offer a first barrier of defense, more work will be needed. We plan to enforce specifications by turning them into dynamic contracts and parametricity by relying on dynamic sealing. We hope that micro-policies can help us implement both contracts and sealing efficiently.

**Micro-policies for C** Micro-policies operate at the lowest machine-code level. While this is appropriate for devising secure C compilers, we also want our secure Low* to C compiler to directly make use of micro-policies in order to efficiently enforce the high-level abstractions of Low*. Moreover, we want a general solution that is not tied to our compilation chain, but instead allows arbitrary programs in C to benefit from efficient programmable tag-based monitoring. Exposing micro-policies in C and then translating them down is challenging, because the structure of programs in these languages is different than that of machine code. We will extend the semantics of C with support for tag-based reference monitoring. These high-level micro-policies will be written in rule-based domain-specific languages (DSLs) inspired by our rule format for micro-policies monitoring machine code [20, 23, 54]. Some parts of the micro-policy DSLs for C and machine code will be similar: for instance, we want a simple way to define the structure of tags using algebraic datatypes, sets, and maps. The kinds of tags differ from level to level though: at the machine code level we have register, program counter, and memory tags, while in C we could replace register tags with value and procedure tags. The way tags are checked and propagated also differs significantly between levels. At the machine-code level, propagation is done via rules that are invoked on each instruction, while in C we have many different operations that can be monitored, e.g., primitive operations, function calls and returns etc. Moreover, the tags of C values could be propagated automatically as values are copied around, without needing to write explicit rules for that. Finally, we want to automatically translate C micro-policies to machine-code ones.

**Secure micro-policy composition** Our secure compilation chains require composing many different micro-policies. For instance, we need to simultaneously enforce isolation of mutual distrustful components and memory safety for some of the components. Recent microarchitectural optimizations enable us to efficiently enforce multiple micro-policies simultaneously [54], by taking tags to be touples, where each tag component is handled by a different sub-policy. Yet composing isolation and memory safety is non-trivial, since each of them has its own view on memory, and a naive composition would be dysfunctional, for instance dynamically allocating in the memory of the wrong component. While this problem can be fixed by changing the code of the composed micro-policies, the bigger conceptual difficulty is in composing specifications and security proofs. Secure composition principles are badly needed, since verifying each composed micro-policy from scratch does not scale. Secure composition is, however, very difficult to achieve in our setting, because micro-policies can directly influence the monitored code by answering to direct calls and by raising catchable exceptions, and thus one micro-policy’s observable behavior can break the other micro-policy’s guarantees. We will study several techniques for composing micro-policy specifications and proofs, with the composite policies needed by our secure compilation chains as motivating examples. First, we will investigate layering micro-policies, by choosing an order among them and constructing a sequence of abstract machines, each of which “virtualizes” the tagging mechanisms in the hardware so that further micro-policies can be implemented on top. We will then use ideas from monad transformers and algebraic effects to allow the micro-policies to be verified separately and layered in any order. Finally, we will investigate forms of composition in which each micro-policy specifies how its tags should be affected by the interactions of the other policies with the monitored code.

**Preserving Confidentiality and Hypersafety** It would be interesting to extend our secure compartmentalizing compilation criterion and enforcement mechanisms [7] from robustly preserving safety to confidentiality and hypersafety [8]. For this we need to control the flow of information at the target level—e.g., by restricting direct reads and read capabilities, cleaning registers, etc. This becomes very challenging though, in a realistic attacker model in which low-level contexts can observe time. While at first we could assume that low-level contexts cannot exploit such side-channels, an interesting challenge will be to try to extend our enforcement to also protect against timing side-channels. In this context, we could investigate preserving various K-Safety Hyperproperties such as nonmalleable information flow control [41], timing-sensitive noninterference [111], and cryptographic “constant time” [34] (i.e., secret independent timing).

**Research Group, Collaborators, and Community** A problem of this size cannot be solved by a single person. I will continue to use my ERC and other grants to fund PhD students and PostDocs to work with me on various pieces. I will further rely on my current collaborations (described in Form 1) and try to establish new ones where meaningful. For instance, I will try to establish a network of French researchers interested in various approaches to making compilers like CompCert more secure, including for instance Xavier Leroy (Collège de France); Frederic Besson, Sandrine Blazy, Thomas Jensen, and David Pichardie (Inria Rennes); as well as Benjamin Grégoire and Tamara Tezk (Inria Sophia Antipolis). Finally, I will continue my effort to build a broader community around secure compilation, by organizing international workshops (like PriSC) and meetings (like Dagstuhl Seminar 20201), and by teaching and trying to build reusable teaching materials.


