Research Statement
Joshua Gancher

Modern software is exceedingly complex. As this complexity grows and permeates our digital lives, we need to find ways of maintaining *trust* in software systems: trust that the system is running *correctly*, and trust that the system is behaving *securely*. In my research, I advocate for using *formal verification* to increase our trust in these systems. Using formal verification, one writes mathematically precise proofs which demonstrate with certainty that software is correct and secure before deployment.

Unfortunately, there is a common belief that formal verification is not very useful due to it being hard to use, or resulting in software with poor performance. My goal is to change this belief via new techniques that broaden the scope of formal verification to new classes of performant software systems with highly automated, scalable proofs. To reason about software in its full complexity, I leverage my expertise in programming languages to develop *formal*, *foundational* proof techniques that enable easily reasoning about systems using novel abstractions.

Formal Techniques for Cryptographic Security. Thus far, my main focus has been developing new techniques for verifying that cryptographic protocols are secure. Ensuring security for cryptographic protocols is critical, as virtually all forms of modern digital communication rely on them to guarantee security in an adversarial environment. On the other hand, cryptographic protocols are very *brittle*: a single protocol flaw — far too easy to accidentally introduce — can topple the security of the entire system.

Verified implementations of low-level cryptographic algorithms are already starting to become adopted in practice, with verified elliptic curves now running in Chrome, Firefox, and the Linux kernel. However, these verification efforts only target bugs in small, well-defined cryptographic algorithms. Errors in high-level protocol *logic* — allowing adversaries to steal secret data and forge bogus messages — are still lurking in protocols and their implementations. To prevent these kinds of bugs, we need to scale up our verification infrastructure to entire protocols.

How to obtain verified, realistic versions of protocols, fast enough to be used in practice, is still largely an open question. I plan on changing this. By fluently speaking the languages of both formal verification and cryptography, my work creates new, useful insights at the intersection of these two disciplines that will enable the creation of performant, verified protocol implementations.

In addition to the creation of new verification pipelines, I create programming language abstractions that enable non-domain experts to harness the power of cryptographic protocols for their own security-critical applications. This is done via *cryptography-aware* programming languages that safely encapsulate the complex security guarantees of cryptographic mechanisms into easy-to-use, concise interfaces.

Verification for Secure Communication Protocols

Formally protecting against protocol-level logic bugs, such as monster-in-the-middle attacks that allow adversaries to pose as honest servers, requires proving cryptographic security of the protocol design. However, tools for ensuring cryptographic security typically only analyze *simplified abstractions* of protocols, missing out on crucial implementation details. While some tools do support security analyses in the presence of concrete implementations, using these tools currently require herculean effort, including intricate on-paper composition theorems and interleaving cryptographic reasoning with implementation details.

To certify real-world, implementable protocol designs, I have developed Owl [2], a new tool for formally verifying secure communication protocols, such as TLS, WireGuard, and SSH. Owl operates in an *end-to-end* manner by cleanly separating the *design* stage of the protocol, where one proves high-level security guarantees, from the *implementation* stage, where the protocol design is compiled into performant, verified software.

Owl guarantees, for the first time, a confluence of three crucial properties for cryptographic protocol verification: *proof modularity*, which allows separate verification efforts to be combined; *proof automation*, which dramatically lessens the burden on the user; and *computational security*, which delivers security guarantees against realistic adversaries who may perform arbitrary computations.

Owl's main technical contribution is the **first type system for cryptographic protocols** that supports **computational security** in the presence of a **wide array of cryptographic mechanisms**, such as authenticated encryption, digital signatures, hash functions, and Diffie-Hellman operations.

Ensuring computational security is the only way to be certain that protocols are secure. Unfortunately, prior provers for computational security either require extensive manual proof effort when applied to protocols, or are limited to non-modular proof methods that perform whole-program analyses, and thus cannot reason about the security of composed protocols separately. In contrast, since our type system has a *once and for all* security guarantee, protocol developers need only write a well-typed program to guarantee security. This is a significantly easier task than mechanizing a cryptographic proof — in particular, type checking is naturally automated, while "raw" cryptographic proofs are not. Using this type system, multiple graduate students with varying levels of cryptographic background have successfully verified a number of protocols, including (simplified versions of) Kerberos and the key exchange protocol from SSH.

The central challenge of Owl's type system is that cryptographic keys in secure protocols must be *well-used*; for example, encryptions under a key K are insecure if K is output in plaintext on the network, or used to encrypt K itself. To enforce that keys are used securely, Owl's type system uses a novel combination of *information flow* and *refinement* types. Notably, the information flow in Owl is *fine-grained*: we do not track information flow between coarse-grained parties (e.g., Alice or Bob) as is typically done in information flow, but between labels associated to individual cryptographic keys themselves.

To support realistic implementations, Owl includes a compiler that lowers the Owl version of the protocol into Rust. Since our IEEE S&P 2023 publication, we have developed a *certifying compiler* from Owl protocols to verified Rust implementations using the Verus framework. Our certifying compiler already guarantees functional correctness and memory safety; we are currently modifying the compiler to guarantee side-channel resistance as well.

Our first target for verified compilation is WireGuard, a secure VPN protocol. We are in the process of using Owl to prove the core protocol of WireGuard secure, and using our certifying compiler to obtain a verified, drop-in replacement that will interoperate with existing implementations. By doing so, we not only produce a verified version of WireGuard that can easily be used by others, but demonstrate that it is reasonable (and hopefully, soon *expected*) to use verified protocol implementations in practice.

Funding Support. As co-PI (along with my postdoc advisor, Bryan Parno), we were awarded an NSF SaTC grant (Award #2224279) for carrying out the Owl project. I contributed a large portion of the technical content for the proposal.

Verification for Distributed Cryptography

While Owl is a promising step towards a future full of verified cryptography, it cannot tell the whole story. Owl is designed to handle protocols which, in a sense, *use* "simple" cryptographic mechanisms (e.g., encryption, signatures, and hash functions) in complex ways; thus, protocols which *construct* new forms of cryptography — requiring subtle probabilistic arguments — are out of scope. In particular, as advanced cryptographic mechanisms such as multi-party computation (MPC) continue to be adopted for cloud computing and machine learning, we will see a growing need for mechanizing their security proofs to ensure the absence of errors in their designs.

We prove these protocols secure in the *simulation-based security* framework, which requires demonstrating *observational equivalences* between protocols and ideal specifications. In essence, two protocols are observationally equivalent when *no* cryptographic adversary can tell the difference between interacting with the first protocol or the second. While this proof strategy is very expressive, observational equivalences are virtually impossible to verify on-paper due to the universal quantification over all possible observers.

Prior mechanized work encodes these observational equivalences through complex *bisimulation* arguments, which relate the states of the two protocols via a hand-written invariant. While expressive, direct bisimulation arguments are extremely difficult to employ: after finding the invariant — which can be highly nontrivial — one must prove the invariant sound. Due to these complexities, prior provers for observational equivalence require thousands of lines of code per proof, even for simple examples.

To simplify proofs of simulation-based security, I have developed IPDL [3] (presented at POPL 2023), a core calculus for cryptographic protocols. Instead of using handwritten bisimulations, IPDL uses an *equational logic* that allows one to progressively *rewrite* the protocol until it matches the specification. Interactively applying rewrites to the protocol is much easier than writing a bisimulation: while bisimulations require whole-program invariants and semantic arguments of soundness, IPDL's rewrite rules are purely local, syntax-guided changes. In effect, IPDL's logic provides a modular, abstract way to easily prove protocols secure.

The central challenge of IPDL is how best to handle the inherent *nondeterminism* present in distributed protocols. If Alice and Bob can both send a message, and the protocol behaves differently based on whose message is sent first, we now have twice the number of cases to consider; thus, without a bisimulation invariant, the proof effort will grow exponentially. My insight which enabled our equational logic is that a large set of cryptographic protocols — including most MPC protocols — *do not rely on the relative ordering of messages* in any substantial way. This property (formalized via a *confluence* property) dramatically simplified the verification task for our protocols of interest.

Using the high-level IPDL logic, we have carried out a number of case studies mechanized in Coq, including the first (and still, only) mechanized proof of observational equivalence between an MPC protocol and its specification. As a point of comparison with tools based on bisimulation, we implemented a modular case study consisting of key establishment using Diffie-Hellman, followed by a one-time-pad message transmission using the established key. Two prior proof efforts for the same protocol took 18,000 and 2,000 lines of code respectively, while our example took only 736 lines for both definitions and proofs.

By making cryptographic proofs easier, techniques inspired from IPDL will not only enable existing protocols to be verified, but will ideally enable cryptographers to develop mechanized versions of their protocols *at design time*, in order to certify their results. While we are not there yet, further developments for easy-to-use verified cryptographic proofs could in principle *accelerate* parts of the design process via computer-assisted proofs.

Programming Languages for Cryptography

In addition to their use in formal verification, programming language techniques have the potential to be transformational in how we *construct* new uses of cryptography. By creating new abstractions — particularly new *type systems* — we can empower developers without cryptographic expertise to safely use cryptography for their own needs. While this is in part the goal of Owl, this new paradigm for programming with cryptography has much broader applications. Indeed, by making advanced cryptographic mechanisms accessible for non-experts, we help enable a new generation of secure software, including privacy-preserving applications for contact tracing and cloud-assisted machine learning.

In pursuit of this goal, I helped develop Viaduct [1] (presented at PLDI 2021), a compiler for secure distributed protocols. To create new protocols using Viaduct, the programmer writes an idealized program that specifies the protocol's overall computational goals. For example, to implement a secure auction, the program takes as input the parties' bids, and outputs who won the auction; by not otherwise using the bids, the program ensures that the bids themselves are otherwise kept secret. In turn, Viaduct automatically compiles this program into a distributed protocol that uses advanced cryptography to realize the desired protocol securely.

The novelty of Viaduct is that it compiles *heterogeneous* protocols which can use multiple cryptographic primitives, such as commitments, MPC, and zero-knowledge proofs. This is very important in practice, as using a variety of cryptographic mechanisms for different purposes is crucial for good performance. By enabling non-expert developers to leverage a variety of cryptographic mechanisms automatically, Viaduct works towards the goal of advanced cryptography becoming deployable at scale.

Without compiler support, correctly choosing which parts of the program may be computed by MPC and which by zero-knowledge requires a great deal of cryptographic knowledge, since both protocols have subtly different security guarantees. Viaduct automates this process using information-flow labels that, in a novel way, capture both secrecy and integrity guarantees for each cryptographic mechanism. At the same time, these information-flow labels provide the programmer a concise but expressive way to specify security policies.

To prove the Viaduct compiler correct, we have formalized parts of the compiler in terms of simulation-based security, similar to what is proven by IPDL. In essence, our proof shows an equivalence between the high-level source program and the compiled protocol: *any* attack leveraged by a low-level adversary against the compiled protocol is equivalent to an attack leveraged by a weaker adversary against the source program. Our work is the first to establish simulation-based security for a general compiler; indeed, we argue that simulation-based security, neglected by the programming languages community, is not only a great fit for compiler correctness, but possible and practical. This work will appear at CSF 2024.

Overall, the Viaduct compiler demonstrates the power of combining programming language techniques with cryptography. By creating a new, highly automated toolchain for developing protocols — and using formal verification techniques to certify this toolchain — we show that advanced cryptographic techniques are within reach for practitioners beyond those with extensive cryptographic expertise.

Future Work

Lightweight Formal Methods for Cryptographic Proofs. While my work aims to achieve fully formal proofs of security, it is currently unrealistic to expect all cryptographic practitioners to adopt formal methods. In parallel with making these tools easier to use, I am interested in preventing protocol logic bugs from the other end, as well:

can we create *lightweight, accessible* formal tools that can be used by cryptographers without formal training in verification?

First, in the short term, I would like to create a *testing framework* for cryptographic proofs — particularly simulation-based ones — that enable cryptographers to find bugs in their proof steps. Some initial work has been done in this direction, but a future effort would focus on creating a highly *usable* language with many cryptographic features built-in; e.g., support for often-used mathematical features in cryptographic protocols, such as groups, lattices, and secret-sharing primitives. Using such a tool, the cryptographer would be encouraged to formalize their constructions using *executable specifications*, and ensure that they have the expected properties via automatic testing (e.g., using a fuzzer).

Second, in the longer term, I believe a *type system* for composing the algorithms used in cryptographic proofs would be enormously helpful. In a vague analogy, Rust has been widely successful in the systems programming community for writing more correct code, simply by ensuring memory safety and allowing the easy use of algebraic data types. I believe a similar story can be told for cryptographic proofs. For example, a type system for so-called "extractors" in proofs of security for zero-knowledge argument systems can reduce the proof bureaucracy a great deal, and likely detect many proof bugs due to misaligned interfaces.

Simulation-Based Security for Implementations. In essence, the simulation-based security framework allows one to relate two programs P and Q with identical compute goals, but different attack surfaces: we say that P *simulates* Q if any (efficient) attack on P can be converted into an equivalent (efficient) attack on Q. While this paradigm has been used mostly to prove security for cryptographic protocols, it can have a much broader impact on the field of security in general. In particular, if we think of P as an implementation and Q as a functional specification, then simulation-based security shows that the implementation "leaks no more" about private data than the specification. This idea has been explored to some degree, but there is still quite a lot to investigate in this space.

For systems which use cryptography, this perspective on secure implementations gives us a clear path towards verifying *computational security for implementations.* The computational security guarantee of Owl, for example, only applies to the protocol specification that the user provides, and not yet to the extracted Rust code. While we are aiming for functional correctness and side-channel resistance in our ongoing work, these two properties alone do not allow us to lower our computational security guarantee onto the implementation.

To obtain computational security for implementations, we can take advantage of simulation being *transitive*: if the implementation simulates the functional specification, and the functional specification simulates its idealization, then the implementation simulates the idealization as well. Carrying out this research program would require a number of technical advances, but has the potential to be paradigm shifting for cryptographic implementations.

Cryptography for Larger Software Systems. Finally, a long-term research goal of mine is to invent new abstractions for composing *security-producing* program modules (e.g., cryptographic protocols, data protection mechanisms, or secure boot routines) with *security-consuming* program modules who rely on the guarantees of the security-producing modules.

For example, how can we *lift* the security guarantees of a verified TLS implementation into the browser, which uses TLS for secure communication? A naive strategy is to embed the browser into the same verification framework as the TLS implementation, but this is unlikely to scale to realistic use cases, since these verification frameworks are typically not fit to reason about large, non-cryptographic programs. Instead, I would like to find new abstractions that allow us to, one day, build browsers using a high-level, scalable systems programming language that is able to reason about about the low-level security guarantees of communication protocols.

References

[1] Coşku Acay, Rolph Recto, Joshua Gancher, Andrew C Myers, and Elaine Shi. Viaduct: an extensible, optimizing compiler for secure distributed programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 740–755, 2021.

[2] Joshua Gancher, Sydney Gibson, Pratap Singh, Samvid Dharanikota, and Bryan Parno. Owl: Compositional verification of security protocols via an information-flow type system. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1130–1147. IEEE Computer Society, 2023.

[3] Joshua Gancher, Kristina Sojakova, Xiong Fan, Elaine Shi, and Greg Morrisett. A core calculus for equational proofs of cryptographic protocols. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023.