ProtoBlocks: Programming language for secure implementation of cryptographic protocols

Vivek Nair
UC Berkeley

William Mullen
UC Berkeley

Ethan Lee
UC Berkeley - CS 263

Abstract

"Memory-safe" languages like Rust have effectively eliminated memory safety vulnerabilities through good programming language design alone. Can the same be done for cryptographic protocol vulnerabilities? We present a formal specification of ProtoBlocks, a "protocol-safe" programming language that enforces best practices for cryptographic protocol implementation. We provide an easy to use custom-purpose syntax for protocol specification that transpiles to standard JavaScript and executes over a blockchain-like data structure with numerous desirable security properties. ProtoBlocks is compatible with all 10 authentication protocols we tested, and is able to automatically rectify vulnerabilities in the Needham-Schroeder and Woo-Lam protocols. The total overhead of using ProtoBlocks over a network is as low as 0.03%-1.17% for protocols using public-key cryptography.

1 Introduction

Prior to the introduction of memory-safe compiled languages like Rust, programmers had two options: write laborious computationally-verifiable proofs of memory safety, or bear the possibility of undetected security vulnerabilities. In practice, most people did the latter, leading application vulnerabilities such as buffer overflows to become widespread; proving all of the properties of a large and complex application borders on infeasible, and most will choose to spend valuable development resources elsewhere. Only when languages like Rust emerged did it become feasible to ensure memory safety without significantly extra effort on the part of the implementer.

Yet today, the state of cryptographic protocol safety is similar to that of memory safety before memory-safe languages, with lengthy computationally-verifiable proofs being the only surefire way to ensure protocol security. Can programming languages do for protocol safety what Rust did for memory safety? In this paper, we present ProtoBlocks, a programming language that aims to answer that very question. Unfortunately, guaranteeing protocol correctness requires a computational model of the "ideal functionality," which is itself a challenging task. Instead, we do the next best thing, which is enforcing best practices for protocol design and implementation which are widely agreed upon and mitigate the vast majority of protocol vulnerabilities in the wild. We use a blockchain-like structure called DataCapsules for executing protocols which provides a number of desirable security properties, and our language compiles down to standard JavaScript code that can be used in unmodified web browsers, Node.js applications, Electron desktop apps, React Native mobile apps, and more. We have performed a detailed evaluation of our solution with respect to performance, security, and compatibility. The source code for our ProtoBlocks library, ProtoBlocks transpiler, DataCapsules library, and experimental test bed are all publicly available and linked at the end of this paper.

1.1 List of Contributions

Our primary contributions are:

- A transpiler for running ProtoBlocks protocols in standard JavaScript, and a method for generating secure ProtoBlocks protocol execution traces using DataCapsules (Section 4).
- The ProtoBlocks language for secure protocol implementation with formally-defined syntax (Section 5) and semantics (Section 6).
- A performance, security, and compatibility evaluation of our solution (Section 8).

An end-to-end example of the ProtoBlocks implementation and execution flow of a simple authentication protocol is provided in Section 7.

2 Background

The vast majority of the literature in this field pertains to the formal verification and provable security of cryptographic protocols [2, 5, 7, 9]. Various techniques exist for manually or automatically generating proofs of protocol properties, but relatively few works concern implementation of the "informal methods" that this paper focuses on. For this, we turn to works like Abadi and Needham's *Prudent Engineering Practice for Cryptographic Protocols* [3], discussed in more detail in Section 3. Major inspirations for our work were the programming languages Prolac, SNAP, and Viaduct, which are the closest existing languages to the ProtoBlocks language we designed. We also took inspiration from memory-safe languages such as Rust [13] and Java [6].

2.1 Other Languages

Prolac [11, 12] is a programming language designed for building network protocols. Its design emphasizes ease of use for developers by providing a number of helpful features for protocol implementation. Prolac provides an excellent template for the syntax and feature set of our language. However, unlike our project, it does not have a security focus and is only concerned with usability.

SNAP [14] is a low-level language for network protocol implementation which, like ProtoBlocks, uses a novel network primitive ("safe and nimble active packets") to achieve new performance and security characteristics. A key difference is that the SNAP protocol does not enforce message ordering, and as a low-level language, SNAP is not compatible with web applications.

Viaduct [4] is a compiler originally proposed by Acay, Recto, and Gancher et al. that translates high level code into secure, efficient, and distributed code based on flags set by the developer. Developers can mark hosts, host authority, and other cryptographic requirements. Additionally, the Viaduct runtime is extensible, allowing for easy implementation of new flags and mechanisms. While there are several similarities to our system, Viaduct still keeps the onus of security on the developer as they must set the flags themselves. Furthermore, there are no tools included that specifically aid the developer in that process. However, Viaduct does automatically select the execution protocol that gets used once a program is compiled and run. Unlike this project, our proposal focuses on how a language can make network protocols inherently secure without developers needing to mark their code, regardless of the actual protocol being used.

2.2 DataCapsules

For our structure, we turn to DataCapsules as originally proposed by Mor and Kubiatowicz in the paper *Global Data Plane: A Widely Distributed Storage and Communication Infrastructure* [15] and later summarized in *Global Data Plane: A Federated Vision for Secure Data in Edge Computing* [16]. While do not leverage the associated storage properties, the inherent structure of DataCapsules fits well with ProtoBlocks. We highlight a few features in later sections.

3 Desired Properties

As a counterpart to formal methods, Martin Abadi's "Prudent Engineering Practice for Cryptographic Protocols" [3] outlines a number of informal design principles which account for the vast majority of vulnerabilities underlying cryptographic protocol implementations. Our approach is to explicitly enforce these principles using a variety of systems, networking and programming language techniques. Specifically:

1. **Explicit Communication:** Every message should explicitly encode its own semantic meaning. Eg. "After receiving bit-pattern P, S sends to A a session key K intended to be good for conversation with B" - message must contain P, S, A, B, and K to be interpreted precisely. Otherwise, an adversary could deceitfully swap one message for another.
2. **Appropriate Action:** Protocol specifications should clearly state the exact circumstances in which a message should be acted on.
3. **Naming:** Every message should explicitly identify the principals it concerns.
4. **Encryption:** Protocol specifications should state why encryption is being used.
    (a) Confidentiality — ensure that only intended recipients know the decryption key.
    (b) Authenticity — ensure that only the intended principal knows the encryption key.
    (c) Binding — if this is the only goal, digital signatures may be sufficient.
    (d) Randomness — if this is the only goal, other one-way functions may suffice.
5. **Signing:** Signing a message does not necessarily imply knowledge of the content. Eg. when signing an encrypted message or a hash of a message, it is not guaranteed that the signer knows (or approves of) what the underlying message is.

6. **Freshness:** Protocol specifications should state why nonces and timestamps are being used.
   (a) Freshness — storage of used/unused nonces must be maintained to prevent reply.
   (b) Temporal Succession — nonces must be sortable to ensure order can be determined.
   (c) Association — encryption or signing must be used to bind nonce to other message parts.
7. **Nonces:** Predictable nonces should be protected from adversaries. If an adversary can determine a nonce used by another party (eg. a known counter), they could try to simulate a challenge and replay the response.
8. **Timestamps:** Timestamps require clock synchronization and thus trustworthy time servers.
9. **Keys:** Recent use of a key does not imply integrity nor recent generation.
10. **Encoding:** Every message should specify its own encoding scheme. Encoding must identify:
   (a) Protocol — which protocol the message belongs to (to prevent cross-protocol attacks)
   (b) Protocol Run ID — which protocol run the message belongs to (to prevent replay attacks)
   (c) Protocol Message ID — which message within a protocol run the message belongs to
11. **Trust:** Protocol specifications should identify and justify trust relationships.

In addition to these principles, we also prioritize three additional key security features for our language.

1. **Integrity:** Any principal should be able to verify the integrity of the protocol. They should be able to detect errors during execution or modifications to previous actions.
2. **Authenticity:** An extension of principles one and three above, only authorized parties should be able to communicate through a specific protocol instance. Unauthorized parties should not be able to directly affect the protocol in any way.
3. **Order:** An extension of principle one and two, the protocol should have a well defined order. Violations of this order should be detectable by any involved party.

What follows is the design for a programming language which aims to automatically encode the above best practices and security guarantees via its syntax and execution environment.

4 Design

We now discuss the design of ProtoBlocks. We break the following into three major components, the language, the transpiler, and the DataCapsule representation.

4.1 ProtoBlocks Language

The ProtoBlocks language provides a convenient way to implement protocols where correct use of the .pb.js syntax is sufficient to ensure a number of desirable security properties. ProtoBlocks protocols are defined as a well-ordered series of steps ([2] appropriate action), each of which has a unique name for both the step itself and all of its expected outputs ([1] explicit communication). Each step explicitly identifies its intended sender and recipients ([3] naming), and when outputs of prior steps are used in later steps, the intended trusted source of the value must be specified ([11] trust). The language also provides secure helper functions for hashing, encryption & decryption ([4] encryption), nonces ([7] nonces), signing & verifying ([5] signing), timestamps ([8] timestamps), and key management ([9] keys). As such, simply using the provided interface enforces good implementation choices.

4.2 ProtoBlocks Transpiler

The goal of the ProtoBlocks transpiler is to transform protocols written in the ProtoBlocks language (.pb.js) into standard JavaScript files (.js) using the ProtoBlocks library, which in turn can be used in servers (node) and a variety of clients (browsers, electron desktop applications, react native mobile applications, etc.). This was accomplished by modifying Babel to support the ProtoBlocks syntax and writing a custom Babel transformer to generate the vanilla JavaScript syntax expected by the ProtoBlocks JavaScript library via direct manipulation of abstract syntax trees (ASTs). This only required defining semantics for two unique keywords (protocol and step), with the rest of the functionality taking advantage of existing JavaScript syntax.

4.3 DataCapsule Protocol Encoding

Underlying our library implementation is a special structure called a DataCapsule. First described in a paper by Nitesh Mor [16] as part of the Global Data Plane (GDP), we adapted them to fit our project. The GDP is an information centric distributed network. It has no central authority, instead routing information between parties using only GDP names. DataCapsules are the base unit of data that is transferred through the GDP.

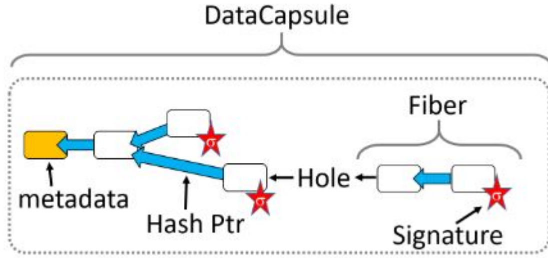


Figure 1: The basic structure of a DataCapsule from [16].

For us, DataCapsules (DCs) are the structure which underpins our JavaScript implementation. A DC is an append-only blockchain of records, each with their own type-agnostic data and hashes. Each record also stores a signed hash of the previous record, creating a well-ordered chain. In our JavaScript DC library, the first record stores protocol metadata while later records represent a specific step. Due to this structure, protocol information and the data associated with each step are bound to specific records and DCs, preventing misuse. The hashes preserve the overall integrity of protocol, ensuring attacks are quickly detected. Thus, the DCs' structure provides a natural match for ProtoBlocks.

4.3.1 Basic Implementation

As originally described and shown in Figure 1, each DataCapsule is defined as a chain of immutable *records*. In their most basic form, these records contain arbitrary data, a hash over the data, and a series of hash pointers. Each record must also specifically contain a signed hash pointer of the previous record, thus creating a well-ordered chain of hashes. The first record in the capsule is a special metadata record that uniquely defines the DataCapsule itself. It contains ownership information, a public key for signing records, and other application specific information. The GDP name of this capsule is defined as the hash of this metadata record. While DataCapsules have several additional features, our implementation is based on this highly flexible basic specification.

4.3.2 Our Version

We make several key changes to the basic DataCapsule implementation for ProtoBlocks. Our DataCapsule library is written in JavaScript, with each capsule representing a single invocation of a protocol and each record representing a single step of that protocol. Like the original version, our DataCapsules are append-only chains of immutable records. However, rather than a hash pointer, we simply store a SHA-256 hash of the previous record in each new record. Furthermore, we make two notable changes to the content of records. First, we encode the protocol's metadata directly into the metadata record of the DataCapsule. This includes the protocol name, version, instance identifier, encoding, timestamp, and principals. Second, we encode the step names and principal executing the step directly into each record. Both changes are for security reasons and are described in the next section. Unlike the DataCapsules described above, our implementation is not dependent on the GDP and is a standalone data structure.

4.3.3 DataCapsule Security Benefits

The base DataCapsule implementation already provides several important security guarantees. The hashes ensure record integrity, allowing any principal or third party to verify the protocol's steps have not been modified. Furthermore, the signed hashes also provide authenticity, as only the principals involved with this protocol invocation have the private signing key. Due to the hash chain structure of the capsule, the records are well-ordered; out of order steps or replay attacks would immediately be detected. Our modifications also add additional protection for protocol specific features. Encoding the protocol metadata directly into the DataCapsule to establish a new invocation covers several of the design principles including naming, timestamps, and encoding. When combined with the additional information added to each record, the DataCapsule also encodes the trust relationship between principals and explicitly defines the communication performed and the order in which it should occur. Through this design, information is precisely bound to records and parties, preventing misuse. Thus, simply by using DataCapsules we gain powerful integrity, authenticity, and provenance guarantees while also improving protocol security through our modifications.

5 Syntax

The ProtoBlocks language contains just two unique keywords, *protocol* and *step*, with the rest of the syntax inherited from JavaScript. We define the syntax of these operations (and thus the language as a whole) as follows:

We begin by defining the `protocol` keyword. It requires a *BindingIdentifier* (protocol name), a list of principals, and the body of the protocol itself (defined below).

ProtocolDeclaration: `protocol` *BindingIdentifier* `[` *ProtocolPrincipalList* `]` { *ProtocolBody* }

We define the *ProtocolPrincipalList* to be a list of principals, each of which is defined by an identifier. Each identifier is made up of a local name used in the *ProtocolBody* and the public key of the associated principal. This is so they are able to later authenticate and write to the DataCapsule.

ProtocolPrincipalList:
  ProtocolPrincipal
  ProtocolPrincipalList **,** *ProtocolPrincipal*

ProtocolPrincipal: *Identifier* (*ProtocolPrincipalInputList*)

ProtocolPrincipalInputList:
  [empty]
  BindingIdentifier
  ProtocolPrincipalInputList **,** *BindingIdentifier*

Finally, we define the *ProtocolBody*. The body is made up of a series of *ProtocolStep* invocations. Each `step` requires a step identifier (or name), a *StepPartiesList*, and the actual action being taken within the step. Note the slightly different definition of the *StepPartiesList* from the *ProtocolPrincipalList*. A *StepPartiesList* requires the protocol developer to indicate the direction of the computation, enforcing who performs the step and who receives the result.

ProtocolBody:
  ProtocolStep
  ProtocolBody ProtocolStepDeclaration

ProtocolStepDeclaration: `step` *Identifier* `[` *StepPartiesList* `]` { *FunctionBody* }

StepPartiesList:
  [empty]
  Identifier
  StepPartiesList `->` *Identifier*

As the syntax demonstrates, we force protocol developers to clearly define each part of their protocol. For example, the *ProtocolPrincipalList* must be defined at the start of the protocol and cannot be changed afterwards. This design thus prevents unauthorized users from adding themselves to the protocol and is enforced through the immutability of DataCapsule records. The *StepPartiesList* functions similarly, though on the granularity of a single step. An example implementation that uses this syntax is presented in Section 7.

6 Semantics

Our language is defined as a superset of the JavaScript language. The complete formalization of the JavaScript language is outside the scope of this project, and there are some difficulties in doing so due to its complex and dynamic nature[19]. As such, we will focus on formalizing the newly introduced semantics of the ProtoBlocks DSL, including the library's underlying use of DataCapsules and built-in cryptographic functions. The rules and notation defined below will also assume that ProtoBlocks programs are defined in isolated *modules*, so that one protocol is defined per program and can be running at a given time.

First, we will provide the program state of a ProtoBlocks program as $\sigma = (\sigma_{\text{JS}}, p, s, \tau)$, with the following definitions:

1. σ_{JS}: The inherited program state and evaluation context of the JavaScript module, including module imports, variable resolution, and built-in functions.
2. $p \in P$: The current named principal executing a protocol. P represents the statically-defined `ProtocolPrincipalList`.
3. $s \in S$: The current step this protocol is in. S represents all `ProtocolStepDeclaration` identifiers and `DONE`.
4. τ: Represents the protocol transcript, as a DataCapsule. The DataCapsule is represented as (i, ℓ_R), where i is an identifier for the DataCapsule and ℓ_R is a pointer to the most recently included block.

We define an operation on σ to update the transcript's DataCapsule with a new block B. This operation adds a block B to τ by adding the hash of the block at ℓ_R to B and then setting ℓ_R to the location of B.

$$\sigma[\tau \mathrel{+}= B]$$

We define a helper function on σ to get the function f associated with a step s as:

$$F(s \in S, \sigma \in \Sigma) = f_s \in \texttt{FunctionExpression}$$

We also define a helper function to get the next step of the protocol:

$$N(s_i \in S) = \begin{cases} s_{i+1}, & \text{if } i < |S| \\ \texttt{DONE}, & \text{otherwise} \end{cases}$$

We can then define rules for the APPEND operation for a DataCapsule, an EVAL operation for a

`ProtocolStepDeclaration`, and a SEND operation for a `ProtocolDeclaration`:

$$\frac{}{\langle \text{APPEND}(\tau, B_*)\,,\, \sigma\,\rangle \Downarrow \sigma\ [\tau\ \texttt{+=}\ B_*]}$$

Given a new block B_*, append it to the program's running transcript.

$$\frac{\dfrac{\langle \text{call}(f, \text{E}_s),\, \sigma\,\rangle \Downarrow \mathbf{v}, \sigma'}{\mathbf{v} \in \texttt{SerializableValue}}}{\langle \text{EVAL}(f, \text{E}_f)\,,\, \sigma\,\rangle \Downarrow v, \sigma'}$$

Use inherited call rule from JS to define a stricter **eval** rule with only serializable values.

$$\frac{\dfrac{\text{F}(\sigma_s) = f}{\langle \text{EVAL}(f, \text{E}_f)\,,\, \sigma\,\rangle \Downarrow \text{B}_*, \sigma'} \quad \langle \text{APPEND}(\tau, B_*)\,,\, \sigma\,\rangle \Downarrow \sigma'' \quad \text{N}(\sigma_s) = s'}{\langle \text{SEND}(\text{E}_p)\,,\, \sigma\,\rangle \Downarrow \sigma''[s = s']}$$

Send a message using the current step of the protocol.

Together, these semantics formally define the underlying functionality of our system. Note that our DataCapsule semantics are only valid for our implementation, and not for the DataCapsules used in the GDP. Making these definitions compatible is a potential extension we discuss in Section 10.

7 Example

We will illustrate the functionality of ProtoBlocks by demonstrating the implementation of the ISO/IEC ISO/IEC 9798-2 [1] Two-Pass Unilateral Authentication over a Cryptographic Check Function (CCF) Protocol, a simple authentication protocol whereby a client authenticates with a server by providing a hash of a shared secret concatenated with a one-time nonce. We also implemented this protocol as part of our performance evaluation, though we leave a discussion of the results to Section 8.

7.1 Security Protocol Notation

1. $\text{B} \rightarrow \text{A}: \text{N}_\text{B}$
2. $\text{A} \rightarrow \text{B}: \text{f}_{\text{KAB}}(\text{N}_\text{B}||\text{B})$
3. B accepts iff CCF matches expected value

We start with the security protocol notation, which contains three steps. In the first step, party B sends party A a one-time nonce (the "challenge"). In the second step, party A sends party B a keyed hash containing the nonce and the identity of B (the "response"). Finally, party B verifies that the response matches the expected value (by locally re-computing the expected hash). This notation defines the specification for the protocol which is then implemented in ProtoBlocks.

7.2 ProtoBlocks Language Implementation (.pb.js)

```
1  protocol ISO_2_Pass_Unilateral_Authentication_Over_CCF [Prover(Secret), Verifier(Secret)] {
2    step Challenge [Verifier -> Prover] {
3      const Nonce = nonce();
4      Prover.send({"Nonce": Nonce});
5    }
6
7    step Response [Prover -> Verifier] {
8      const Hash = hash(Verifier.Challenge.Nonce + Verifier.Id + Prover.Input.Secret);
9      Verifier.send({"Hash": Hash});
10   }
11
12   step Verify [Verifier] {
13     const Hash = hash(Verifier.Challenge.Nonce + Verifier.Id + Verifier.Input.Secret);
14     return (Hash === Prover.Response.Hash);
15   }
16 }
```

Shown above is the encoding of the specified protocol in the ProtoBlocks language. Per the ProtoBlocks syntax requirements, the parties A and B are given explicit roles (*Prover* and *Verifier*) with explicit protocol inputs (*Prover*(*Secret*) and *Verifier*(*Secret*)). Each step is also given a descriptive name (*Challenge*, *Response*, and *Verify*) and an implicit execution order. The steps are further each defined in terms of a specific sender and recipient (*Challenge*[*Verifier*− > *Prover*], *Response*[*Prover*− > *Verifier*], and *Verify*[*Verifier*]). Note that when a value from a previous step is referenced in a later step, the trust assumptions (namely, the intended trusted source of the value) must be explicitly stated (eg. *Verifier.Challenge.Nonce* rather than just *Nonce*). The result of these various design decisions is a ProtoBlocks protocol specification that is very well-defined and leaves little room for ambiguity, leading to an exact one-to-one JavaScript transpilation and later a precisely-defined DataCapsule protocol execution trace.

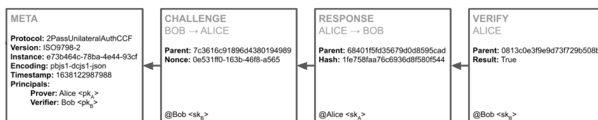
7.3 ProtoBlocks Library Implementation (.js)

```
1  const { Protocol, hash, nonce } = require('../../src')
2
3  /* eslint camelcase: "off" */
4  const ISO_2_Pass_Unilateral_Authentication_Over_CCF = new Protocol({
5    name: 'ISO_2_Pass_Unilateral_Authentication_Over_CCF',
6    principals: [{name: 'Prover', inputs: ['Secret']}, {name: 'Verifier', inputs: ['Secret']}],
7    steps: [{origin: 'Verifier', recipients: ['Prover'], name: 'Challenge', function: async (Prover, Verifier) =>
8      const Nonce = nonce()
9      Prover.send({Nonce: Nonce})
10   }}, {origin: 'Prover', recipients: ['Verifier'], name: 'Response', function: async (Prover, Verifier) => {
11     const Hash = hash(Verifier.Challenge.Nonce + Verifier.Id + Prover.Input.Secret)
12     Verifier.send({Hash: Hash})
13   }}, {origin: 'Verifier', recipients: [], name: 'Verify', function: async (Prover, Verifier) => {
14     const Hash = hash(Verifier.Challenge.Nonce + Verifier.Id + Verifier.Input.Secret)
15     return Hash === Prover.Response.Hash
16   }}]
17 })
18
19 module.exports = ISO_2_Pass_Unilateral_Authentication_Over_CCF
```

Above is the JavaScript output of the ProtoBlocks transpiler when supplied the aforementioned ProtoBlocks language input. The result is a library call to the ProtoBlocks library with a Javascript object specifying the protocol details. As seen here, the ProtoBlocks

library interface is somewhat complicated and is not intended for direct consumption. Therefore, the intended course of action is for protocols to be written in ProtoBlocks code and then transpiled to JavaScript code using the ProtoBlocks library (as this guarantees a well-defined unambiguous protocol with no missing property definitions). This is analagous to React.js, which could theoretically be used directly as a JavaScript library, but is intended for consumption via JSX that is then transpiled into JavaScript using React library calls. In fact, the same transpilation technology (Babel) is used in both React and ProtoBlocks.

7.4 DataCapsule Execution Trace



After compilation and execution, we can view the result in DataCapsule format. On the far left, we see the first record contains the protocol's metadata, clearly defining the details associated with this invocation as well as the principals involved. We then see the three steps of the protocol, each with their own record. Each of these records contains the name of the step, the direction of the operation, the principal performing the step, the data associated with the step itself, and a hash of the previous record. Not pictured are additional hashes over the data and record for further integrity checks. Critical identity information, in this case a secret key, in each step is precisely bound to the relevant party, ensuring only they can correctly add a record to the chain. This trace describes a complete invocation of the 2-Pass Unilateral Authentication Over CCF protocol, with fully defined steps, principals, and order of operations. Using DataCapsules, we can ensure the implementation created by the ProtoBlocks transpiler is cryptographically secure and avoids many vulnerabilities while also following Abadi's design principles.

8 Evaluation

The aim of the ProtoBlocks language is to provide a platform for implementing a wide variety of cryptographic protocols and achieving fast, secure execution of those protocols. Our metrics of success were therefore:

1. **Compatibility** — ProtoBlocks should provide a rich feature set sufficient for implementing a wide variety of existing cryptographic protocols

2. **Security** — ProtoBlocks should automatically detect and mitigate violations of protocol security best practices, making it possible to securely implement protocols that otherwise would otherwise have been vulnerable

3. **Efficiency** — The computational and network overhead of protocols executed by ProtoBlocks should not be cost prohibitive when compared with executing the same protocols in another comparable runtime environment

To that end, we performed a compatibility evaluation, security evaluation, and efficiency evaluation to evaluate the success of ProtoBlocks in these three areas.

8.1 Compatibility Evaluation

To evaluate the compatibility of ProtoBlocks, we chose 10 well-known standardized authentication protocols and attempted to implement and execute them in the ProtoBlocks language. These protocols were:

1. ISO 1 Pass Unilateral Authentication

2. ISO 1 Pass Unilateral Authentication Over CCF

3. ISO 2 Pass Unilateral Authentication Over CCF

4. ISO 3 Pass Mutual Authentication

5. ISO Public Key 1 Pass Unilateral Authentication

6. ISO Public Key 2 Pass Unilateral Authentication

7. Needham Schroeder Symmetric Key Protocol

8. Nonce Return 2 Pass Unilateral Authentication

9. Nonce Return Public Key 2 Pass Unilateral Authentication

10. Woo Lam Mutual Authentication

We began by implementing each protocol in the ProtoBlocks language in close adherence to the original specification. We found that the ProtoBlocks syntax was robust enough to implement all ten protocols. We then ran these ProtoBlocks files through the ProtoBlocks transpiler and were able to successfully generate a JavaScript output for all ten protocols. Lastly, we verified the correctness of the protocols by writing specific unit tests for each protocol using the JavaScript implementation and were able to confirm the desired functionality was present for every protocol. From this, we can conclude that ProtoBlocks achieves widespread compatibility with a large number of standard authentication protocols.

We also performed a compatibility analysis of our DataCapsule implementation. In all ten cases, we did not find any issues with the DataCapsule traces that would

indicate incompatibility. DataCapsule records are flexible enough that, as long as the data is serializable, they could hold anything required by these protocols.

We chose to focus on authentication protocols because they can typically be specified in a fixed number of sequential steps and because it easy to understand the correctness of these protocols in terms of correctly identifying a party. This does mean that some protocols with a variable number of steps, including certain consensus protocols, are less suitable for implementation in ProtoBlocks in its current form.

8.2 Security Evaluation

After verifying the compatibility of ProtoBlocks with numerous authentication protocols, we wanted to confirm that ProtoBlocks provides a tangible security benefit. We did this by focusing on three specific authentication protocols with well-known vulnerabilities: the Needham–Schroeder Symmetric Key Protocol, the Needham–Schroeder Public-Key Protocol, and the Woo-Lam Public-Key Protocol. In each case, we implemented and ran the protocols in ProtoBlocks and evaluated the resulting Data Capsule trace to verify (on paper) that the resulting structure should not be vulnerable to the attack present in the original protocol. We willl summarize why this is the case for each of the three protocols below:

8.2.1 Needham–Schroeder Symmetric Key Protocol

The standalone protocol is vulnerable to a replay attack in which a shared key K_{AB} between parties A and B can be used by a separate party C to incorrectly authenticate as B in a later protocol instance, The ProtoBlocks compiler was able to automatically fix this by binding K_{AB} to a single protocol instance identifier in the metadata block, which must be signed by parties A and B. The immutability of DataCapsule records thwarts a replay attack by C.

8.2.2 Needham–Schroeder Public-Key Protocol

The standalone protocol is vulnerable to a classic man-in-the-middle attack in which A is executing the protocol with C and C is executing the protocol with B, allowing C to authenticate as B. ProtoBlocks was able to automatically fix by this issue by binding the established key K_{AB} to the identity (and public keys) of principals A and B in the metadata block, thwarting a man-in-the-middle attack by C. Furthermore, since the identities defined in the metadata block are later used in each `step`, only these principals are able to append the correct records to the DataCapsule. This prevents any later interference attempts by C.

8.2.3 Woo-Lam Public-Key Protocol

The standalone protocol is vulnerable to a replay attack because a nonce (N_A) was not bound to its issuer (A) but was only guaranteed to be unique with respect to that particular issuer. ProtoBlocks was able to automatically fix this by binding N_A to the identity (and public key) of A in the metadata block; it also ensures nonces are globally unique.

Clearly, ProtoBlocks effectively mitigates potential security vulnerabilities simply by design. It enforces Abadi's principles while also leveraging the powerful security properties of DataCapsules without requiring major changes to the protocols themselves. In fact, in all three of these cases we implemented the protocol as closely to their original specification as possible. Thus, ProtoBlocks can protect against a wide range of common protocol attacks simply by design.

8.3 Performance Evaluation

Protocol	Overhead w/o Network	Overhead w/ Network
ISO_1_Pass_Unilateral_Authentication_Over_CCF	753.33%	1729.41%
ISO_2_Pass_Unilateral_Authentication_Over_CCF	2920.00%	0.03%
ISO_1_Pass_Unilateral_Authentication	708.70%	912.90%
Nonce_Return_2_Pass_Unilateral_Authentication	1183.33%	0.03%
ISO_Public_Key_1_Pass_Unilateral_Authentication	43.57%	48.61%
ISO_Public_Key_2_Pass_Unilateral_Authentication	13.68%	1.17%
Nonce_Return_Public_Key_2_Pass_Unilateral_Authentication	790.20%	0.03%

We evaluated ProtoBlocks' performance by implementing seven well-known authentication protocols. In this suite of tests, we wanted to evaluate the impact of the additional steps ProtoBlocks introduces on the overall time to complete the protocol. As shown above, our results vary substantially across protocols and tests. We found the main source of overhead to be the required public key cryptography associated with signing each record in the DataCapsule.

For protocols that already use public key cryptography, such as ISO Public Key 2-Pass Unilateral Authentication, ProtoBlocks does not add substantial overhead. These protocols already incur the cost of PKC, thus adding an additional signature does not affect the overall runtime. This is especially true when the additional networking overhead is introduced, lowering the added ProtoBlocks overhead to only 0% - 1%. We saw similar behavior in protocols with multiple rounds of symmetric key cryptography. Again, the costs already incurred from the cryptographic operations and networking mostly eclipsed the cost associated with using ProtoBlocks.

Unfortunately, protocols that only use a single round of symmetric key cryptography incurred major performance penalties, often around 700%. These protocols,

especially when performed locally, were substantially slowed by the hashing and signing introduced by our DataCapsule implementation. Furthermore, even adding networking was not enough to recover our performance loss. The overhead of PKC simply eclipses the protocol's normal latency. Some of these costs can be reduced through optimizations to the cryptographic operations, but we leave this to future work.

Today, many modern protocols use multiple symmetric key cryptography invocations or PKC. Furthermore, they also often occur over the network. For those that do not, they are often so fast already that our increased overhead translates into little impact on users. As a result, we feel the minimal performance impact of ProtoBlocks combined with its powerful security guarantees make it a compelling tool for implementing and testing protocols.

9 Related Work

In addition to the work discussed in Section 2, we also analyzed several other papers for our project. While not all are directly related, they did influence the approach we took when designing ProtoBlocks.

9.1 Formal Methods

Abadi and Fournet's *Mobile Values, New Names, and Secure Communication* [2] extends pi calculus with value passing, primitive functions, and equations among terms to analyze security protocols. They call this extension applied pi calculus. They also demonstrate the ability to formally represent several examples, including message authentication codes (MACs) and hashing algorithms.

Dynamic Tags for Security Protocols [5] by Arapinis et al. describes a way to analyze how security protocols perform over an unbounded number of sessions. Specifically, the authors divide a protocol into communication events and status events. They then prove a single session of the protocol is secure. By applying a transformation to the formal representation of the single session, they can obtain a version of the same protocol that is decidably secure over an arbitrary number of sessions.

In *A Computationally Sound Mechanized Prover for Security Protocols* [7], Blanchet proposes a tool that automatically produces the proof for various cryptographic protocols in the computational model of formal analysis. The tool breaks a protocol down into a sequence of games in which the probability of attacker success is probabilistically negligible. It can also determine the number of sessions for which a protocol is secure.

Deciding security properties for cryptographic protocols. Application to key cycles [9] by Comon-Lundh et al. explores how to determine whether security protocols function under a bounded number of sessions is NP-complete. The authors do so by reducing systems to a set of solved forms which represent a finite group of traces that verify specific security properties. The authors use this to prove that deciding whether a key cycle exists is NP-complete as well as the decidability of authentication properties and protocols with timestamps.

9.2 Security

The fields of usable security and HCI, and a number of results therein [22, 21, 20, 8], are also relevant to our project, as usability is a core advantage of our "informal methods" approach over formal methods. We assert that the reason tools allowing protocol security to be formally verified have not eliminated protocol vulnerabilities is the significant burden they place on developers to not only implement a protocol but also to prove its security. A number of relevant works in this field are those that specifically study the usability and comprehensibility of particular algorithms from a development perspective. For example, Ongaro et al. [18] discusses the ease of use of consensus algorithms. Similarly, Naylor et al.'s *The Cost of the "S" in HTTPS* [17] analyzes large ISP datasets to quantify the cost of widespread SSL/TLS adoption. The authors weigh these costs against the security benefits of these technologies and discuss proposals to optimize the cost/benefit ratio.

10 Future Work

While we have demonstrated an effective use of the ProtoBlocks language with 10 different protocols already, some extensions can be made to the language to support a wider body of protocols in the future. For example, the step block syntax currently supports protocols with sequential steps, but additional language features could allow for the expression of protocols where some groups of steps occur in parallel. Another example is the static number of steps that are specified in the protocol; to support a variable number of steps or marking some steps as optional, the step block may need to encode additional information about its optionality or number of rounds.

At the library level, more work can be done to strengthen the security guarantees provided by ProtoBlocks. One technique that could supplement the library is the use of taint tracking on inputs to the protocol and the usage of those inputs in step-provided code. Upon identifying variables at risk of leaking sensitive information or being modified, the library might automatically encrypt the records containing these variables to mitigate these types of attacks. DataCapsules already support encrypted data, so adding a taint tracking system would be a powerful extension.

We could also implement strict information flow control at the library level by labelling principals of the protocol with varying levels of security. The library could then perform assertions during step execution to prevent information from flowing from high- to low-security principals by default. This addition could extend the work already done in systems such as HiStar [23] or Taintdroid [10].

Finally, one other potential extension is to use ProtoBlocks to natively implement various protocols within the GDP. Currently, our DataCapsule implementation is not connected to the original GDP, as it requires specially designed clients for access. However, using ProtoBlocks to convert common protocols into a DataCapsule format would offer a powerful translation layer to widen GDP access. The ProtoBlocks language already converts the protocol into a DataCapsule. Rather than two parties accessing the capsule over a standard network connection, it would instead be managed by the GDP. This way, a wide diversity of systems and clients could gain the benefits of using the GDP without needing complex overhauls to their core networking infrastructure.

11 Conclusion

In this paper, we have presented ProtoBlocks, a programming language for the secure implementation of cryptographic protocols. Our language automatically enforces the protocol security best practices described by Abadi & Needham and is therefore capable of avoiding the vast majority of protocol security vulnerabilities that occur in practice. We introduced a custom ProtoBlocks syntax and provided formal specifications of the language syntax and semantics as well as a well-tested reference implementation. We then presented a ProtoBlocks transpiler which can transform ProtoBlocks code into normal JavaScript code that can run in unmodified web browsers, Node.js applications, etc. We further presented the ProtoBlocks JavaScript library, which executes protocols specified in the ProtoBlocks format using a DataCapsules encoding with several desirable security properties. Lastly, we performed a thorough evaluation which demonstrated wide compatibility for authentication protocols, promising security results for three chosen case studies, and an extremely low performance overhead for protocols using public-key cryptography. Overall, we hope that the key ideas of (1) using custom-purpose languages for protocol implementation and (2) using blockchain-like structures to encode protocol traces receive further study in the future.

11.1 Acknowledgements

Thanks to Deevashwer Rathee and Julien Piet for their advice and contributions in the early stages of this project, and to professors Natacha Crooks, Alvin Cheung, John Kubiatowicz, and Dawn Song for their advice and support.

11.2 Artifacts

The reader is welcomed to explore and verify our results via the following publicly-available repositories:

ProtoBlocks Library

- https://github.com/ProtoBlocks/ProtoBlocks
- https://www.npmjs.com/package/protoblocks

ProtoBlocks Transpiler

- https://github.com/ProtoBlocks/Transpile
- https://github.com/ethanlee16/babel
- https://www.npmjs.com/package/@protoblocks/babel-parser
- https://www.npmjs.com/package/@protoblocks/babel-core
- https://www.npmjs.com/package/@protoblocks/babel-traverse
- https://www.npmjs.com/package/@protoblocks/babel-types

DataCapsule Library

- https://github.com/GlobalDataPlane/DataCapsuleJS
- https://www.npmjs.com/package/datacapsulejs

ProtoBlocks Experiment

- https://github.com/ProtoBlocks/Evaluation

References

[1] 14:00-17:00. ISO/IEC 9798-2:2019.

[2] Abadi, M., and Fournet, C. Mobile values, new names, and secure communication. *ACM Sigplan Notices 36*, 3 (2001), 104–115.

[3] Abadi, M., and Needham, R. Prudent engineering practice for cryptographic protocols. *IEEE transactions on Software Engineering 22*, 1 (1996), 6–15.

[4] Acay, C., Recto, R., Gancher, J., Myers, A. C., and Shi, E. Viaduct: an extensible, optimizing compiler for secure distributed programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (2021), pp. 740–755.

[5] ARAPINIS, M., DELAUNE, S., AND KREMER, S. Dynamic tags for security protocols. *arXiv preprint arXiv:1405.2738* (2014).

[6] ARNOLD, K., GOSLING, J., AND HOLMES, D. *The Java programming language*. Addison Wesley Professional, 2005.

[7] BLANCHET, B. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing 5*, 4 (2008), 193–207.

[8] CLARK, S., GOODSPEED, T., METZGER, P., WASSERMAN, Z., XU, K., AND BLAZE, M. Why (special agent) johnny (still) can't encrypt: A security analysis of the apco project 25 two-way radio system. In *USENIX Security Symposium* (2011), vol. 2011, pp. 8–12.

[9] COMON-LUNDH, H., CORTIER, V., AND ZĂLINESCU, E. Deciding security properties for cryptographic protocols. application to key cycles. *ACM Transactions on Computational Logic (TOCL) 11*, 2 (2010), 1–42.

[10] ENCK, W., GILBERT, P., HAN, S., TENDULKAR, V., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS) 32*, 2 (2014), 1–29.

[11] KOHLER, E. *Prolac–a language for protocol compilation*. PhD thesis, Massachusetts Institute of Technology, 1998.

[12] KOHLER, E., KAASHOEK, M. F., AND MONTGOMERY, D. R. A readable tcp in the prolac protocol language. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication* (1999), pp. 3–13.

[13] MATSAKIS, N. D., AND KLOCK II, F. S. The rust language. In *ACM SIGAda Ada Letters* (2014), vol. 34, ACM, pp. 103–104.

[14] MOORE, J., HICKS, M., AND NETTLES, S. Practical programmable packets. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)* (2001), vol. 1, pp. 41–50 vol.1.

[15] MOR, N. *Global Data Plane: A Widely Distributed Storage and Communication Infrastructure*. PhD thesis, University of California, Berkeley, 2019.

[16] MOR, N., PRATT, R., ALLMAN, E., LUTZ, K., AND KUBIATOWICZ, J. Global data plane: A federated vision for secure data in edge computing. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* (2019), IEEE, pp. 1652–1663.

[17] NAYLOR, D., FINAMORE, A., LEONTIADIS, I., GRUNENBERGER, Y., MELLIA, M., MUNAFÒ, M., PAPAGIANNAKI, K., AND STEENKISTE, P. The cost of the "s" in https. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2014), CoNEXT '14, Association for Computing Machinery, p. 133–140.

[18] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (USA, 2014), USENIX ATC'14, USENIX Association, p. 305–320.

[19] PARK, D., STEFĂNESCU, A., AND ROŞU, G. Kjs: A complete formal semantics of javascript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2015), pp. 346–356.

[20] RUOTI, S., ANDERSEN, J., ZAPPALA, D., AND SEAMONS, K. Why johnny still, still can't encrypt: Evaluating the usability of a modern pgp client. *arXiv preprint arXiv:1510.08555* (2015).

[21] SHENG, S., BRODERICK, L., KORANDA, C. A., AND HYLAND, J. J. Why johnny still can't encrypt: evaluating the usability of email encryption software. In *Symposium On Usable Privacy and Security* (2006), ACM, pp. 3–4.

[22] WHITTEN, A., AND TYGAR, J. D. Why johnny can't encrypt: A usability evaluation of PGP 5.0. In *8th USENIX Security Symposium (USENIX Security 99)* (Washington, D.C., Aug. 1999), USENIX Association.

[23] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIERES, D. Making information flow explicit in histar. *Communications of the ACM 54*, 11 (2011), 93–101.