Verifpal User Manual

FIRST EDITION

Nadim Kobeissi

Published by Symbolic Software

Copyright © 2026 Nadim Kobeissi. All Rights Reserved. "Verifpal" and the Verifpal mascot are registered trademarks of Nadim Kobeissi.

Verifpal User Manual is licensed under Creative Commons Attribution NonCommercial NoDerivatives 4.0 International License (CC BY-NC-ND 4.0); you may not use or share this material except in compliance with its license. Verifpal User Manual acts as supporting material for the Verifpal Software, which is a distinct artifact from the Verifpal User Manual. Verifpal Software is licensed under the GNU General Public License, Version 3 (GPLv3); you may not use or share Verifpal Software except in compliance with its license.

- *CC BY-NC-ND 4.0:* `https://creativecommons.org/licenses/by-nc-nd/4.0/`
- *GPLv3:* `https://www.gnu.org/licenses/gpl-3.0.en.html`

Typeset on April 20, 2026

Acknowledgments

Verifpal is fundamentally inspired by Prof. Bruno Blanchet's decades-long legacy of seminal work on formal verification.[1] In a more spiritual sense, Verifpal is also inspired by *Undertale* by Toby Fox.

This work would not have been possible without the generous support of the NLNet Foundation, which recognized Verifpal's potential and was a quintessential partner in making it a reality. Funding was provided through the *NGI0 Privacy Enhancing Technologies Fund*, a fund established by NLnet with financial support from the European Commission's *Next Generation Internet* program, under the aegis of DG Communications Networks, Content and Technology under grant agreement №825310.

I would also like to thank my students Georgio Nicolas and Sasha Lapiha, as well as Vlad Antipin, for their feedback on earlier versions of the Verifpal User Manual.

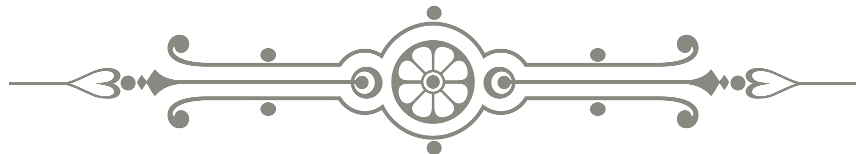
Finally, I am profoundly grateful for the talented artists at Collateral Damage Studios, who worked closely with me over a period of three months to draw the Verifpal manga and illustrations. Their passion for their work allowed Verifpal to exist exactly as I envisioned, a hero to inspire anyone who wishes to learn formal verification.

[1]Note that this should not be interpreted as any sort of endorsement of Verifpal by Prof. Bruno Blanchet.

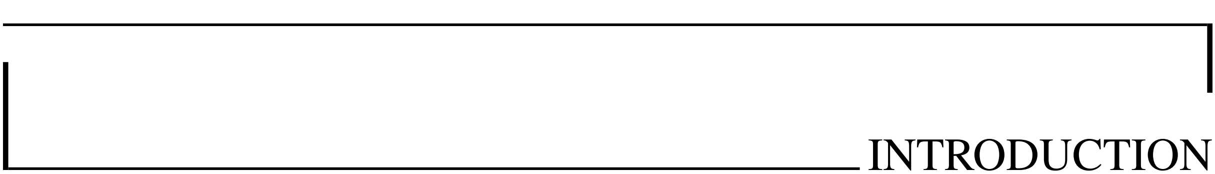
Dedication

Karthikeyan Bhargavan, Bruno Blanchet, Antoine Delignat-Lavaud, Graham Steel and Harry Halpin are heroes because they disregarded who I was in favor of who I could become. The well of gratitude that I have for them cannot be filled within my lifetime.

To my students.



8aad60363031ce951081da9eba8f4f24



INTRODUCTION

Verifpal is software for verifying the security of cryptographic protocol designs. Such protocols are all around us: whenever you log into an online account or perform a banking transaction, you use HTTPS and its underlying TLS protocol. Whenever you send a message over WhatsApp, you use the Signal secure messaging protocol.

These protocols need to take care of some serious cryptographic responsibilities, which we call *security properties*: TLS needs to ensure that your password is transmitted to Microsoft Outlook without being readable by any middleman. It also needs to give you a way to make sure that you're sending it to Outlook.com and not some impersonator. We call these security properties *confidentiality* and *authentication*, respectively.

It's important to be able to verify whether TLS and Signal actually achieve these properties. Imagine, for example, if protocol flaws were found in prototypes of TLS that would allow for weaker security [1]. Most communications over the Web would be affected, potentially leading to the compromise of immeasurable amounts of confidential information from all facets of life.

However, protocols can get pretty complex, and reasoning about their security properties can leave you lost in a labyrinth of logical representations and eventualities. That is why, more than a decade ago, major strides began to happen in *automated formal verification*. Software such as ProVerif [2] and Tamarin [3] began to appear, making it possible for researchers to write models in which they formally describe the protocol they want to verify and the security properties it's supposed to satisfy[2]. The protocols that ProVerif and Tamarin have to handle can get pretty intense: Figure 2 shows a slightly simplified execution of the Signal protocol in which Alice initiates a session with Bob, sends a message, and then receives a reply from Bob. Go ahead and flip to the Appendix at the end of this manual so you can take a look at the figure and see what I mean.

[2]ProVerif, Tamarin and also Verifpal verify protocols in the *symbolic model*, also known as the Dolev-Yao model [4]. In the symbolic model, cryptographic operations are treated as ideal "black-box" functions: the only way to decrypt is to possess the correct key, and hash functions are unbreakable one-way functions. This abstracts away concrete implementation details such as key lengths or block cipher modes and focuses analysis on the *logical structure* of the protocol. Computational protocol verifiers, such as CryptoVerif [5], take a different approach by reasoning about concrete cryptographic properties, but the computational model and its differences compared to the symbolic model [6] are not within the purview of this manual.

Intimidated? Perfect. You're reading the right manual: I created Verifpal exactly to make it easy for everyone to understand how to verify cryptographic protocols, even if you have little prior experience with how protocols work or how they are supposed to be designed.

A major focus of my Ph.D. studies was modeling the latest Web protocols in ProVerif, including TLS and Signal. I targeted not only confidentiality and authentication as security goals, but more advanced properties such as *forward secrecy* and *post-compromise security* [7]. The former asks the question: *"does stealing Alice's device allow the thief to decrypt messages she sent in the past?"*, while the latter asks whether the protocol can *recover* its security guarantees after a compromise — that is, whether messages sent sufficiently long after a key compromise are once again protected.

ProVerif's analysis is currently considered state-of-the-art. The software has been under development for close to two decades, and is capable of verification scenarios and queries that are far more advanced than what Verifpal can accomplish today. However, I quickly came to understand that some of its design decisions would leave it at a disadvantage with a wider audience who deserves to have a starting chance at formal verification, but does not have access to the specific background or culture from which ProVerif emerged.

For example, ProVerif more or less assumes an idiomatic understanding of the ML syntax tradition (which inspired ProVerif's modeling language, the *"applied pi-calculus"* [8]). It also expects the user to intuitively reason about protocols as Horn clauses [9] that appear over the network, and not as, say, messages between explicit principals such as Alice or Bob, which is far more likely to be the natural way most people think about secure protocols. Furthermore, whenever ProVerif finds an attack, it outputs long, complex *attack traces* which can sometimes require something of an archaeological expedition in order to read and understand.

Verifpal's design methodology is the inverse of the one usually seen in formal verification research: in designing Verifpal, I wanted to focus on the user first, and on state-of-the-art formal verification last. Yes, you read that correctly: *last*. Making advanced formal verification my final goal does not mean that I don't intend to get to it: it's rather that I will only allow increases in verification capability and features if and only if I know for sure that they can reach the user intuitively and without harming the accessibility of the formal verification experience.

All recent research in this area, without exception, has so far proceeded in the opposite direction: creating more impressive formalization and theoretical advancement first, and worry about making them actually deployable last [10]. While this approach is certainly defensible and, in academic research, sometimes strictly necessary, it has led to (in my opinion) some amount of wasted effort and opportunity.

In designing Verifpal, I focused on ensuring that it offers the following:

- *An intuitive language for modeling protocols.* Verifpal's internal logic still relies on the deconstruction and reconstruction of abstract terms, similar to ProVerif. However, it reasons about the protocol model with *explicit principals*: Alice and Bob exist, they have independent states, they know certain values and perform operations with cryptographic primitives. They send messages to each other over the network, and so on. The Verifpal language is meant to illustrate protocols close to how one may describe them in an informal conversation, while still being precise and expressive enough for formal modeling.

- *Modeling that avoids user error.* Verifpal does not allow users to define their own cryptographic primitives. Instead, it comes with built-in cryptographic functions: `ENC` and `DEC` representing encryption and decryption, `AEAD_ENC` and `AEAD_DEC` representing authenticated encryption and decryption, `SIGN` representing asymmetric primitives, etc. — this is meant to remove the potential for users to define fundamental cryptographic operations incorrectly[3]. Verifpal also adopts a global name-space for all constants and does not allow constants to be redefined or assigned to one another. This enforces models that are clean and easy to follow.

- *Analysis output that's easy to understand.* ProVerif provides attack traces that illustrate a deduction using session-tagged values in a chain of Horn clause deconstructions. As Verifpal is analyzing a model, it outputs notes on which values it is able to deconstruct, conceive of, or reconstruct. When a contradiction is found for a query, the result is related in a readable format that ties the attack to a real-world scenario. This is done by using terminology to indicate how the attack could have been possible, such as through a mayor-in-the-middle on ephemeral keys.

- *Integration with the developer's workflow.* Verifpal comes with a Visual Studio Code extension that offers syntax highlighting, automatic formatting, live analysis, diagram visualizations and much more, allowing developers to obtain insights on their model as they are writing it.

It is worth noting that Verifpal is a protocol *analysis* tool, not a proof-producing verifier: when Verifpal finds a contradiction to a query, it has identified a genuine attack scenario — any reported attack is a real logical consequence of the model. However, when no contradiction is found, this means that Verifpal's bounded analysis was unable to find one within its search space — it does not constitute a formal proof that no attack exists. This is an important asymmetry: Verifpal's attack-finding is *sound* (reported attacks are real), but its verification is not *complete* (the absence of a reported attack does not guarantee security). Chapter 4 discusses Verifpal's analysis methodology and the precise boundaries of its results in more detail.

When you use Verifpal, I expect you to be able to model protocols using a language that immediately makes sense to you. I expect you to receive insight that is immediately understandable, so long as you know what a hash function is, what encryption is, how Diffie-Hellman and signatures work, and a few other core details. I expect Verifpal to give everyone the means to not only experiment with modeling protocols, but also to gain legitimate and novel insights through their modeling.

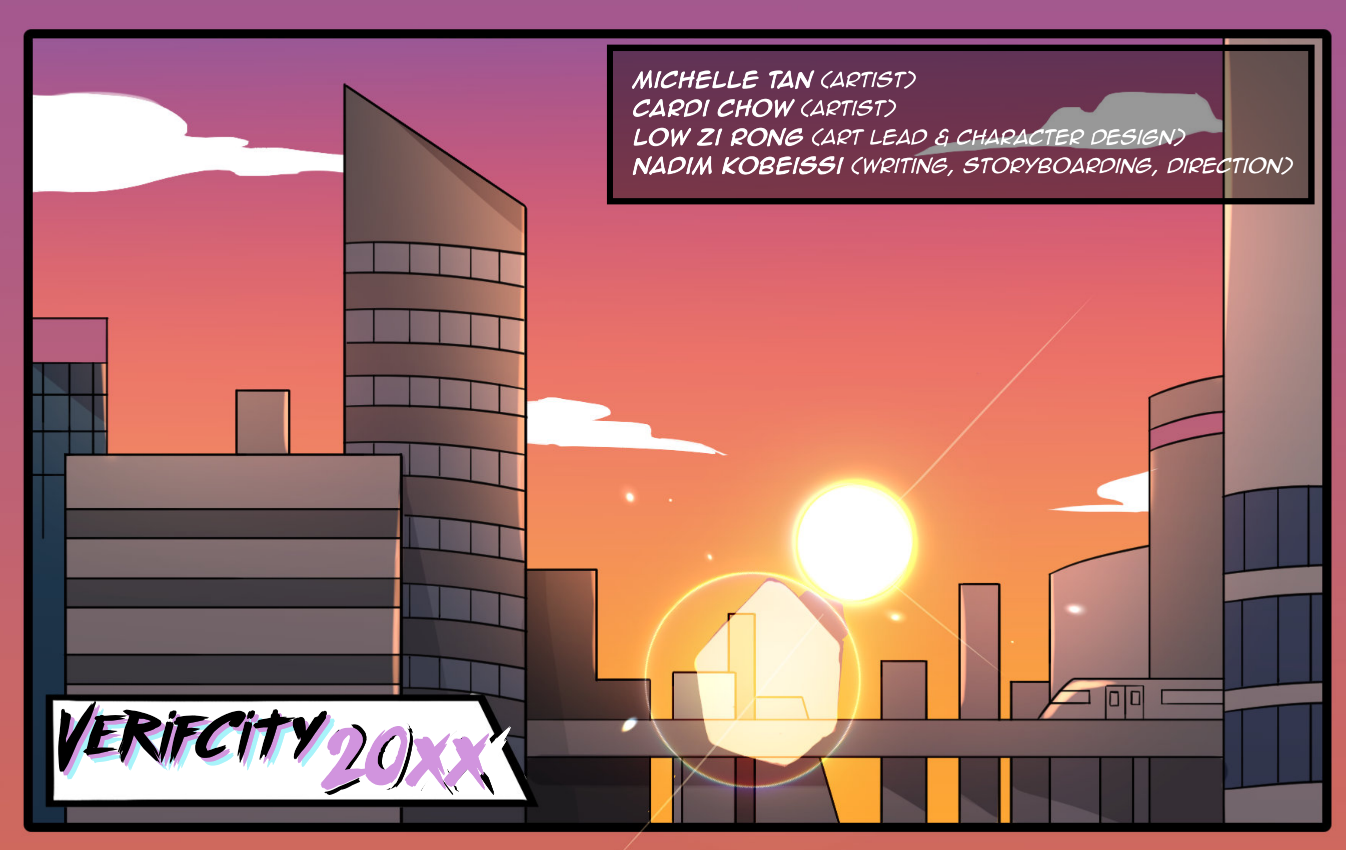
For the true beginner, I suggest, as a companion to this manual, *Serious Cryptography* by Jean-Philippe Aumasson, or *Real-World Cryptography* by David Wong. Both are wonderful books that can help you understand the basics.

Like all heroes, Verifpal thrives in the midst of adventure. What protocols will you and Verifpal venture within? What interesting discoveries will you make?

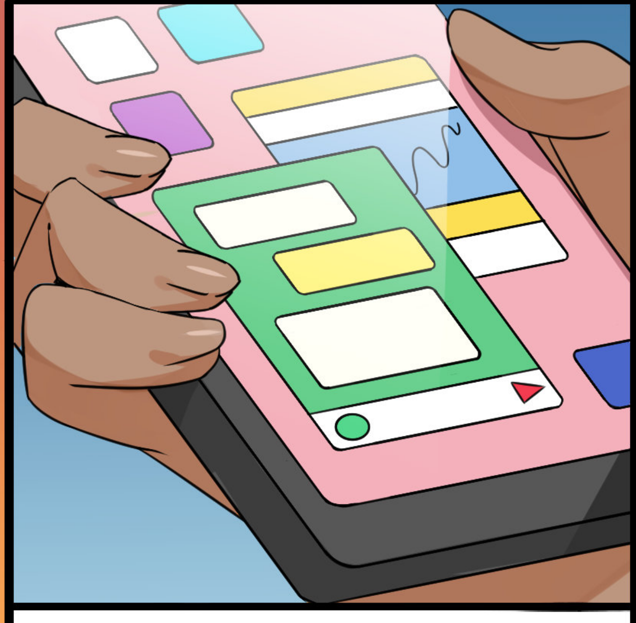
[3]This is an example of how Verifpal fundamentally diverges from ProVerif when it comes to certain goals — its focus on ease of use will allow ProVerif, for the foreseeable future, to provide more elaborate models due to, for example, support for user-defined primitives.

Nadim Kobeissi

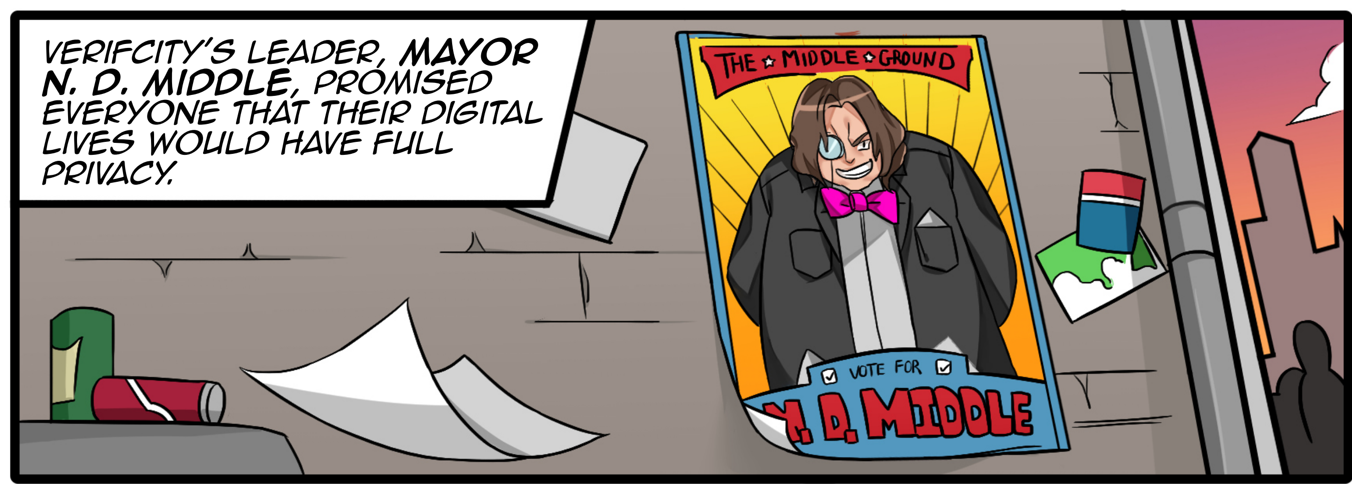
July 27, 2019

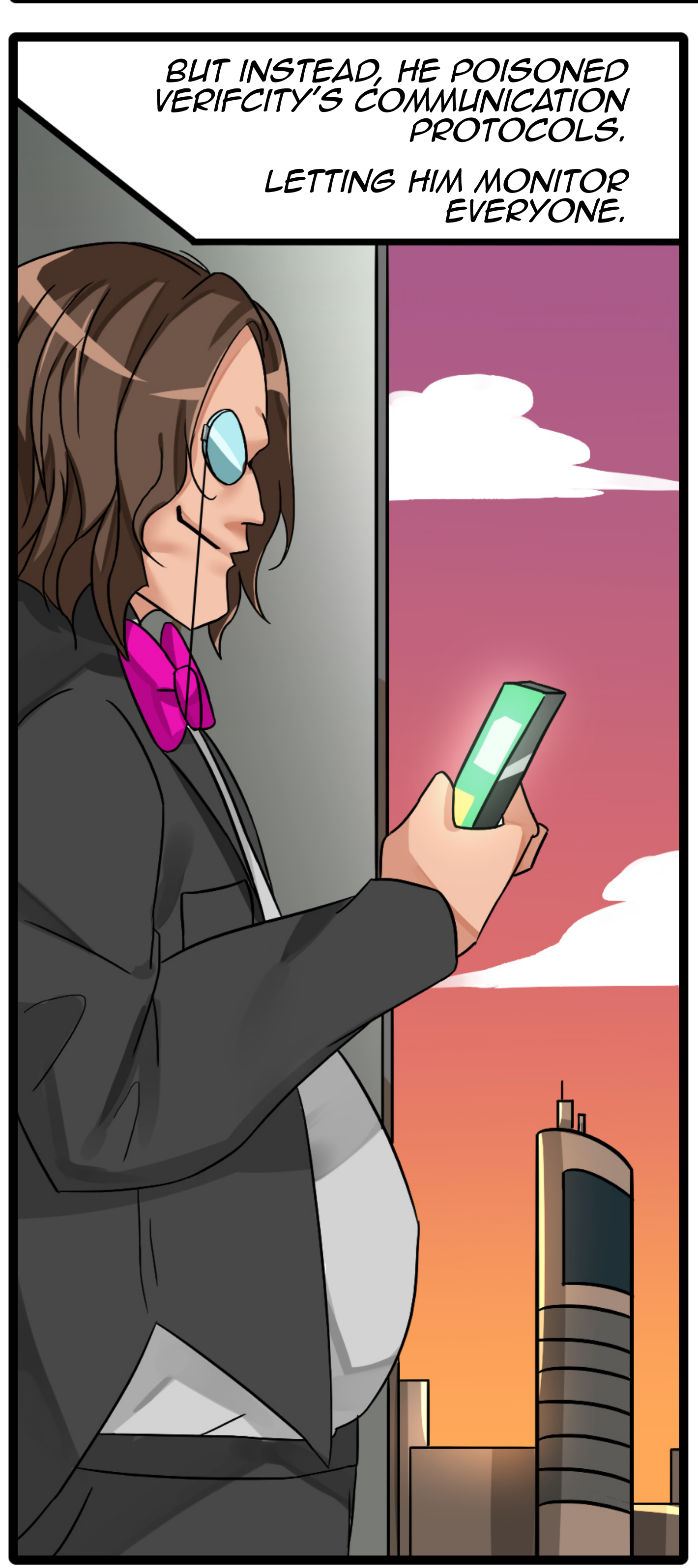

MICHELLE TAN (ARTIST)
CARDI CHOW (ARTIST)
LOW ZI RONG (ART LEAD & CHARACTER DESIGN)
NADIM KOBEISSI (WRITING, STORYBOARDING, DIRECTION)
VERIFCITY 20XX

A NEW ERA IS DAWNING ON VERIFCITY.
AN ERA WHERE EVERYTHING HAPPENS THROUGH SMARTPHONES.
OK, sure!
Cool! See ya soon!


FRIENDSHIPS, BREAKUPS, ELECTIONS, CHECKUPS, BANK ACCOUNTS...
EVERYTHING GOES THROUGH A SINGLE WINDOW INTO PEOPLE'S LIVES.

VERIFCITY

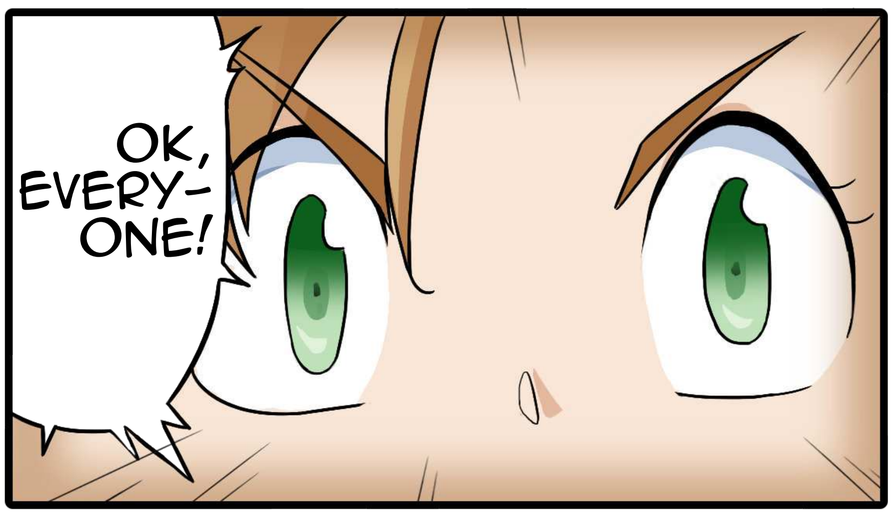

VERIFCITY'S LEADER, MAYOR N. D. MIDDLE, PROMISED EVERYONE THAT THEIR DIGITAL LIVES WOULD HAVE FULL PRIVACY.
THE MIDDLE GROUND
VOTE FOR
N. D. MIDDLE


BUT INSTEAD, HE POISONED VERIFCITY'S COMMUNICATION PROTOCOLS.
LETTING HIM MONITOR EVERYONE.

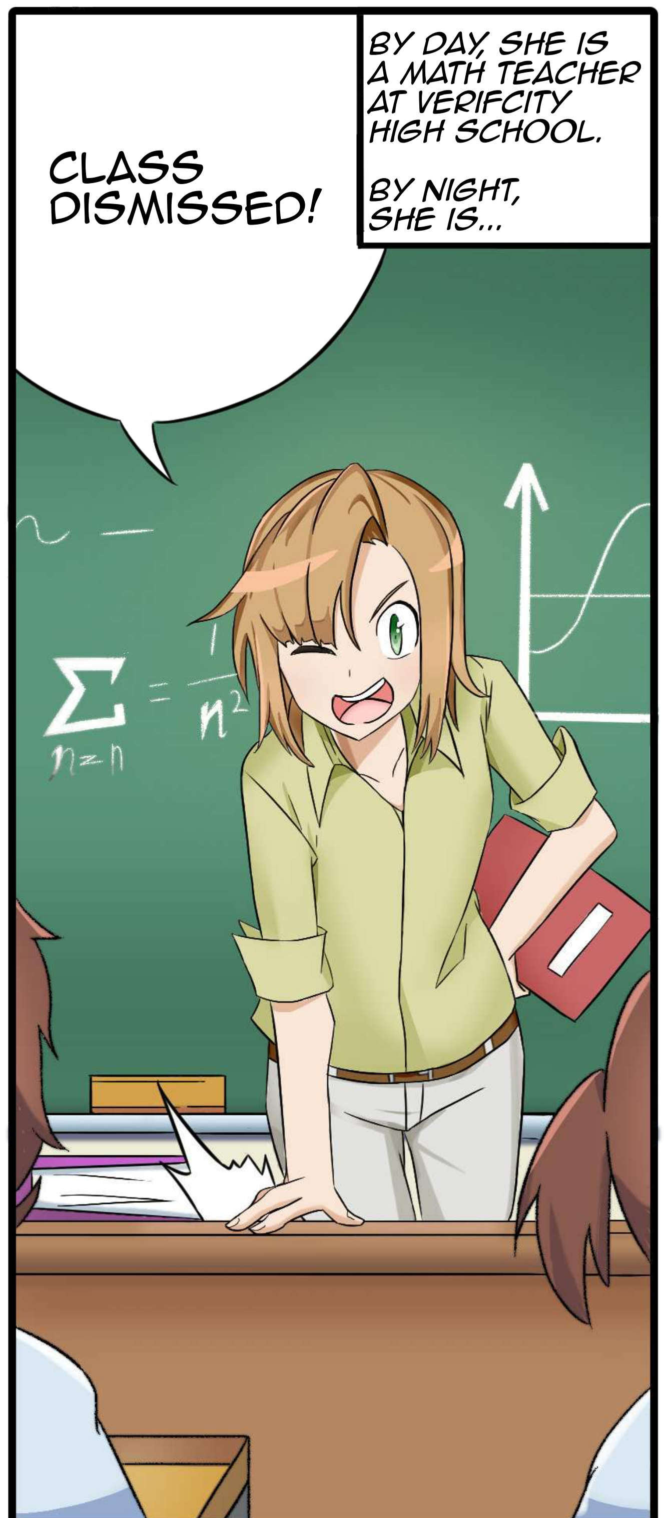
can't wait 2 see u
Me too, sweethea
Let's kee
a secret
You make me feel alive.
EVER SINCE HIS ELECTION, PEOPLE'S LIVES HAVE BEEN AT RISK OF EXPOSURE.
NOBODY KNOWS WHERE IT'S COMING FROM,OR WHEN IT WILL STOP.

WELL, ALMOST NOBODY.

RIIIING!

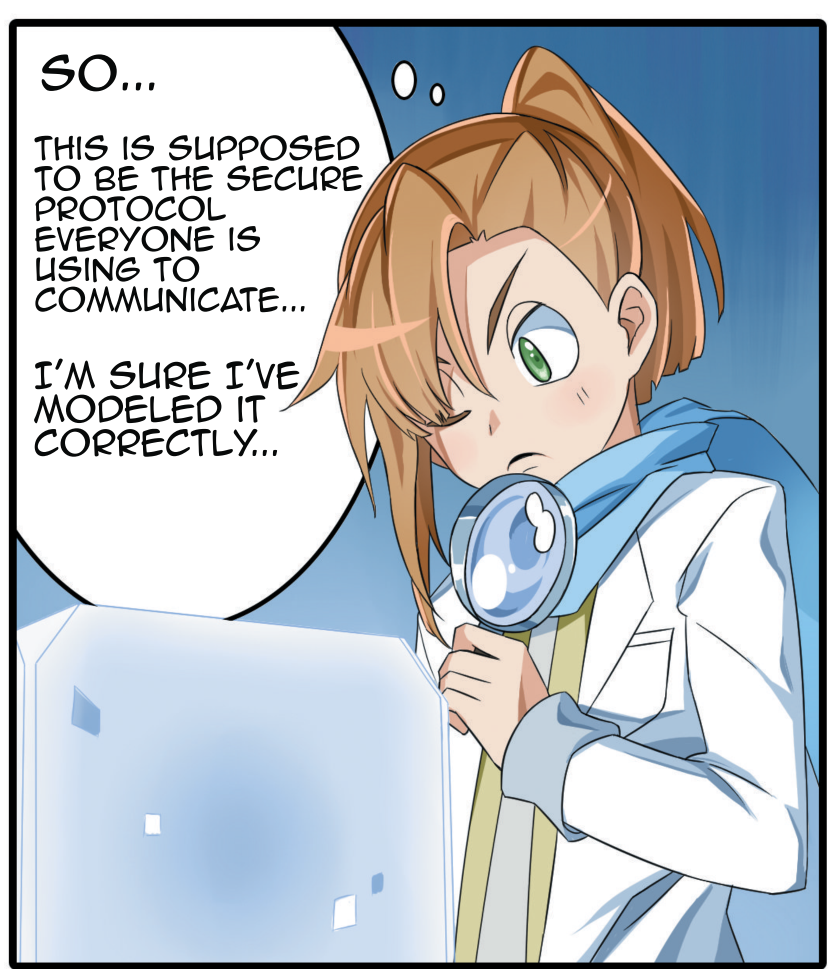


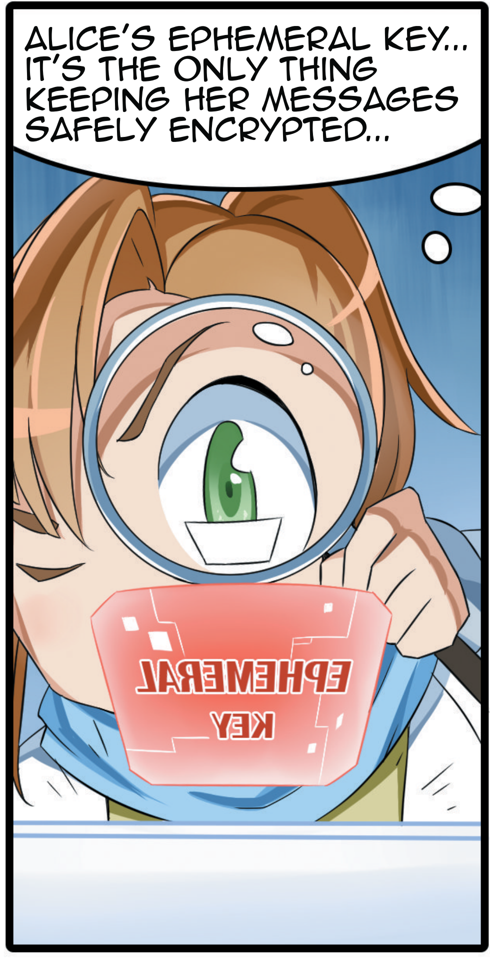
OK, EVERY-ONE!



CLASS DISMISSED!
BY DAY, SHE IS A MATH TEACHER AT VERIFCITY HIGH SCHOOL.
BY NIGHT, SHE IS...

VERIFPAL, THE HERO WHO CAN EXPOSE THE MAYOR'S HIDDEN TYRANNY.
VERIFLAB





YEARS EARLIER...

"BUT SOMETHING'S NOT RIGHT..."

NO, VERIFPAL. A COMPROMISED EPHEMERAL KEY CAN STILL MEAN TROUBLE.

BUT PROVERIF-SAMA! THE LONG-TERM KEYS HAVE MUTUAL AUTHENTICATION!

SO, HAVE YOU CHECKED IF THE AUTHEN-TICATION IS CHAINED DOWN PAST THE KEY EXCHANGE MES-SAGES?!

!

THAT'S RIGHT!

NORMALLY, MESSAGE KEYS ARE DERIVED NOT ONLY FROM ALICE'S EPHEMERAL KEY, BUT ALSO FROM A ROOT KEY...

Alice's Ephemeral Key

Master Secret

HKDF

HKDF

Root Key

HKDF

THIS IS WHAT TIES THE MESSAGES TO ALICE AND BOB'S IDENTITIES...

ENCRYPT

BUT... THE ROOT KEY IS NEVER GETTING MIXED INTO ALICE'S NEW MESSAGE ENCRYPTION KEY!

WHICH MEANS...

IF AN ACTIVE ATTACKER REPLACES THE EPHEMERAL KEY,

THE ENTIRE SESSION GETS COMPROMISED!

THEY CAN READ EVERYTHING!

CRASH

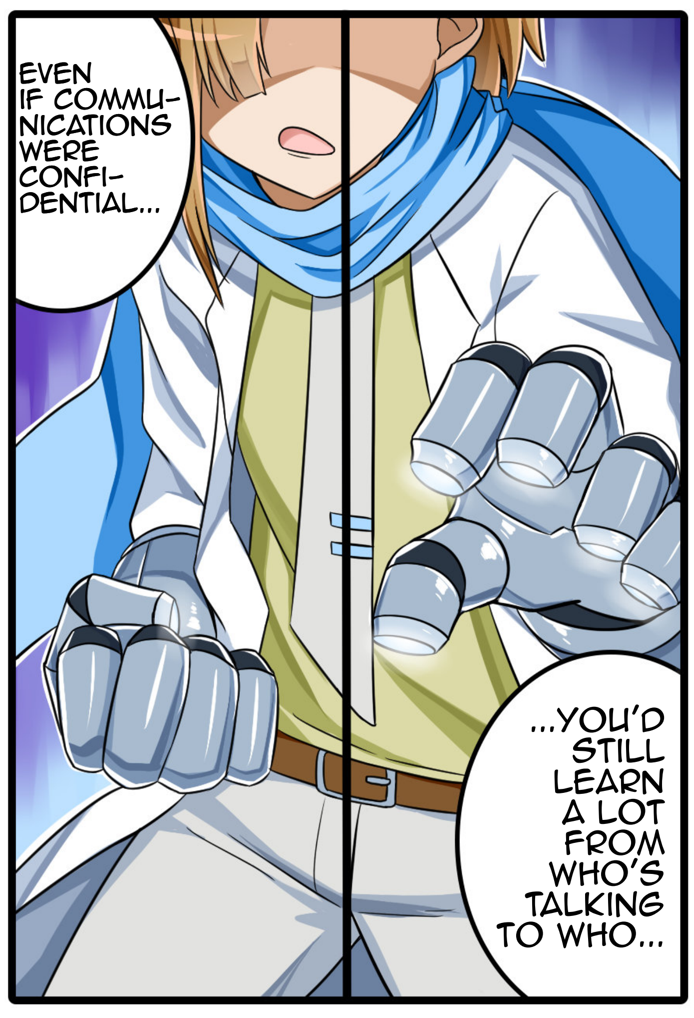
EXCELLENT DEDUCTION AS ALWAYS, VERIFGAL!

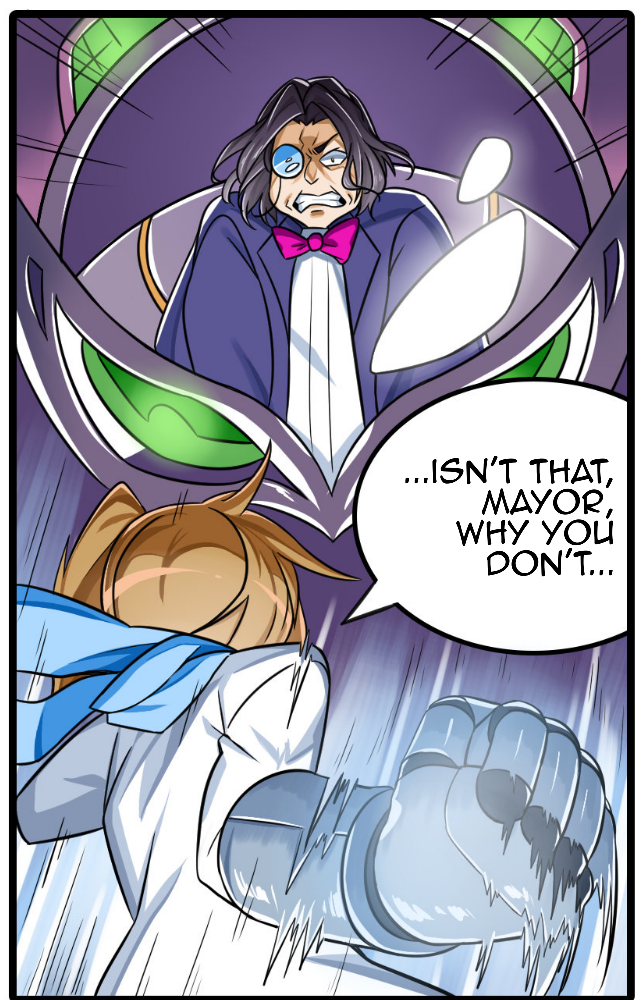
MAYOR N. D. MIDDLE!

HA!

INDEED, THE KEY EXCHANGE IS QUITE WORTHLESS...
BUT DID YOU THINK THAT WAS THE ONLY ACE UP MY SLEEVE?!

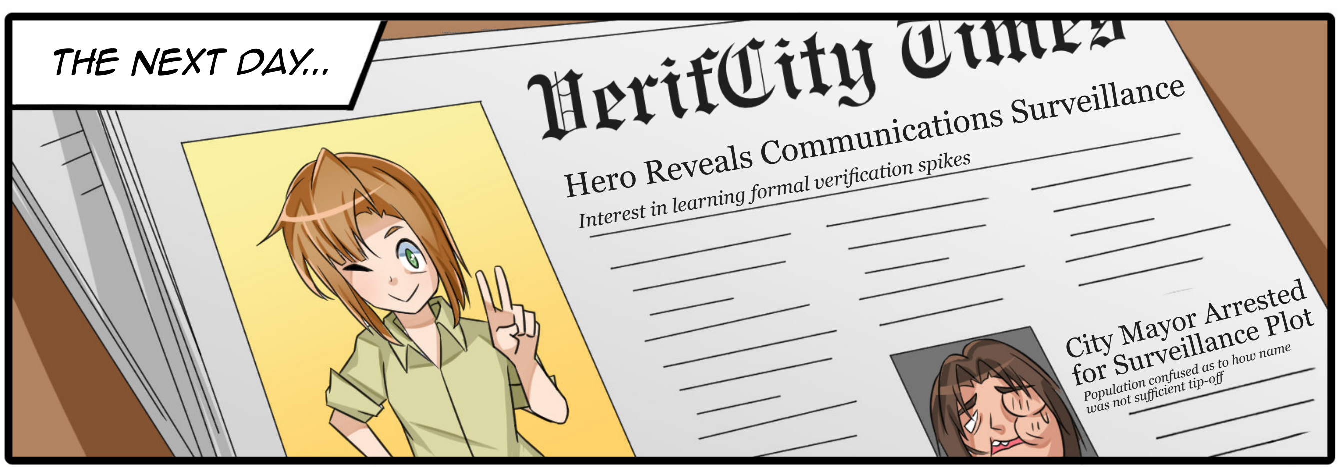
NO, I DIDN'T...

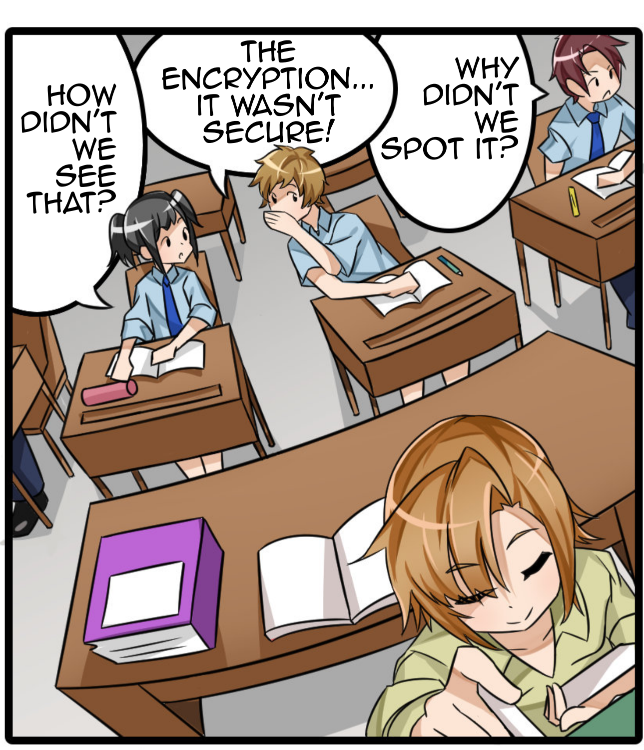


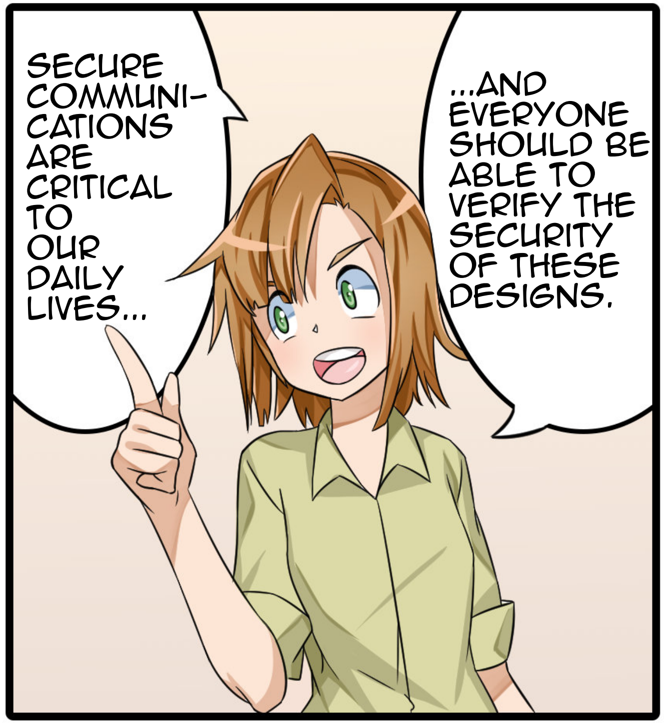


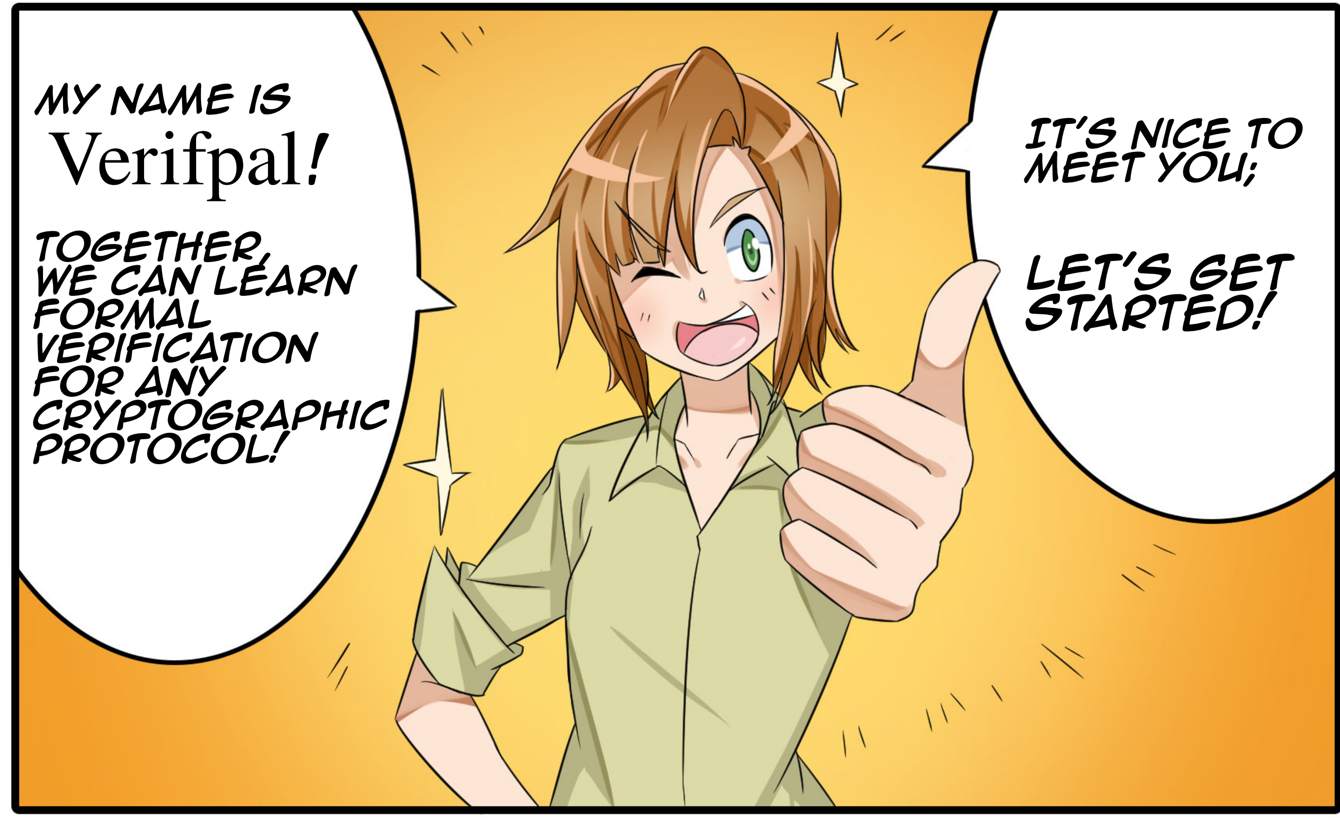
...ENCRYPT THE INITIATOR'S LONG-TERM PUBLIC KEY?!

IM-IMPOSSIBLE!!!


THE NEXT DAY...
VerifCity Times
Hero Reveals Communications Surveillance
Interest in learning formal verification spikes
City Mayor Arrested for Surveillance Plot
Population confused as to how name was not sufficient tip-off


HOW DIDN'T WE SEE THAT?
THE ENCRYPTION... IT WASN'T SECURE!
WHY DIDN'T WE SPOT IT?


SECURE COMMUNI-CATIONS ARE CRITICAL TO OUR DAILY LIVES...
...AND EVERYONE SHOULD BE ABLE TO VERIFY THE SECURITY OF THESE DESIGNS.


MY NAME IS Verifpal!
TOGETHER, WE CAN LEARN FORMAL VERIFICATION FOR ANY CRYPTOGRAPHIC PROTOCOL!
IT'S NICE TO MEET YOU;
LET'S GET STARTED!

CONTENTS

I Getting Started with Verifpal **1**

1 Setting Up Verifpal **2**

1.1 Downloading Verifpal 2

1.2 Installing Verifpal Manually 3

1.3 Running Verifpal 3

1.4 Updating Verifpal 4

1.5 Compiling Verifpal from Source Code 4

1.6 Verifpal for Visual Studio Code 4

1.7 Sharing Verifpal Models 5

1.8 News and Discussion 6

2 The Verifpal Language **7**

2.1 Declaring the Attacker 7

2.2 Principals 8

2.3 Fundamental Types in Verifpal 8

2.4 Messages 15

2.5 Phases 16

2.6 Queries 17
2.7 Comments 17
2.8 A Simple Complete Example 17

3 Protocols and Queries in Verifpal 19

3.1 Use Cases and Security Goals 20
3.2 Queries 21
3.3 Passive and Active Attackers 27
3.4 Understanding Verification Results 29
3.5 Modeling a Challenge-Response Protocol 29

4 Analysis in Verifpal 32

4.1 The Verification Pipeline 32
4.2 Analysis Methodology 33
4.3 Skeleton-Based Injection 36
4.4 Preventing State Space Explosion 36
4.5 Password Extraction 37
4.6 Soundness and Completeness 38

II Protocol Examples in Verifpal 42

5 Secure Messaging with Signal 43

5.1 Security Goals 43
5.2 Principals 44
5.3 Queries and Analysis 48

6 Gossip with Scuttlebutt 51

6.1 Security Goals 51

6.2 Principals 51

6.3 Queries and Analysis 54

7 Contact Tracing with DP-3T **57**

7.1 Security Goals 57

7.2 Modeling DP-3T 57

7.3 Queries 62

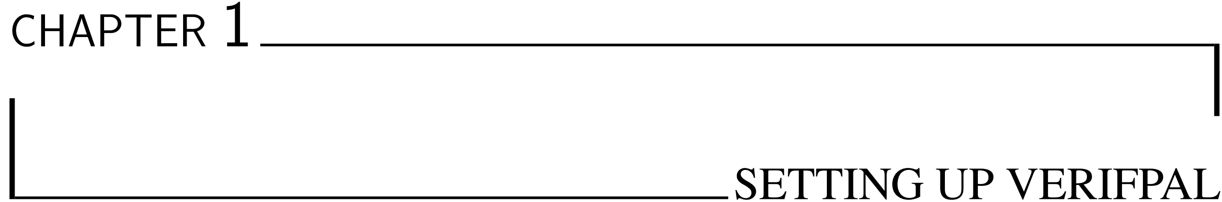
Bibliography **65**

Appendix **67**

Notes **71**

PART I

Getting Started with Verifpal



CHAPTER 1 SETTING UP VERIFPAL

Setting up Verifpal on your computer is easy, and should not take more than five minutes regardless of your computer or operating system.

1.1 DOWNLOADING VERIFPAL

Verifpal is available for Microsoft Windows, Linux, macOS and FreeBSD. In order to download Verifpal, simply visit `https://verifpal.com` and download the latest version for your computer.

On Windows, Verifpal is available via the Scoop[1] package manager. On Linux and macOS, Verifpal is available via the Homebrew[2] package manager. Instructions for installing Verifpal through Scoop or Homebrew are listed on the Verifpal website's *Software & Media* page: `https://verifpal.com/software`.

Installing Verifpal via package manager is the best way to keep Verifpal automatically up to date.

As a reminder, Verifpal is free and open source software, available under the GNU General Public License Version 3. To learn more about your rights and obligations under this license, please review `https://www.gnu.org/licenses/gpl-3.0.en.html`.

[1] `https://scoop.sh`
[2] `https://brew.sh`

Verifpal is Beta Software

Verifpal now benefits from a higher level of assurance due to the formalization of its syntax, semantics and analysis methodology, both by hand and using the Coq theorem prover. However, it remains classified as beta software due to its relatively young age, especially when compared to similar tools, such as ProVerif [2], that have been in development for more than twenty years.

```
~
❯ verifpal
Verifpal 0.10.4 - https://verifpal.com
  Warning • Verifpal is experimental software.
verify    [file]: Load and analyze Verifpal model from file.
implement [file]: Implement Verifpal model in Go.
pretty    [file]: Pretty-print Verifpal model.
help:             Show this help text.

~
❯
```

Figure 1.1: Verifpal in macOS.

1.2 INSTALLING VERIFPAL MANUALLY

Verifpal is a command-line application. There is no specific procedure for installing it, although adding it to your system's command `PATH` may make it easier to use.

1.2.1 *Windows*

Running `verifpal.exe` directly by double-clicking it from the Explorer won't do anything — you will have to open a command-line terminal. The quickest way to do so would be to type ⊞ + R to launch the Run dialog box, and then to type `cmd`.

Once inside a terminal, `cd` to the folder containing `verifpal.exe`.

1.2.2 *Linux, macOS and FreeBSD*

Simply open a terminal and `cd` to the folder containing `verifpal`. You may also wish to copy Verifpal (using `cp`) to a folder within your system `PATH`. For example:

```
cp verifpal /usr/local/bin/verifpal
```

This will allow you to type `verifpal` from any folder on your system in order to quickly run Verifpal.

1.3 RUNNING VERIFPAL

Running `verifpal` should give the output seen in Figure 1.1. Some options are shown:

- `verify`: takes as a parameter a `.vp` file, containing a model for verification.
- `pretty`: outputs a pretty-printed version of the provided `.vp` model, potentially making it more readable.

Once you are able to obtain the output shown in Figure 1.1, you have confirmed that Verifpal is ready to go on your computer.

1.4 UPDATING VERIFPAL

Verifpal software is under continuous development. It is recommended that you periodically visit `https://verifpal.com` to check if a new version of Verifpal is released. New versions can bring improved performance, bug fixes and even new features.

To check which version of Verifpal you have installed, simply run `verifpal` with no arguments. For example, the output shown in Figure 1.1 indicates the installed Verifpal version. Once you've downloaded an updated copy of Verifpal, running and installing it should be possible using the same process described within this chapter.

This Verifpal User Manual that you are reading now will also be updated in time. To check which edition of the manual you currently have, simply consult the manual's cover. Newer editions may be available on the Verifpal website at `https://verifpal.com`.

1.5 COMPILING VERIFPAL FROM SOURCE CODE

You may choose to compile Verifpal from source code instead of downloading a pre-compiled release, although note that there is no significant advantage or difference between downloading a pre-compiled Verifpal binary and compiling your own. Links to the Verifpal source code repository are available on the Verifpal website.

Installing Git. You must have the Git distributed version control system installed on your computer in order to download a copy of the Verifpal source code repository. Please Review the *Git Getting Started*[3] instructions in order to understand how to best install Git for your computer and operating system.

Installing Rust. You must have the Rust programming language toolchain installed in order to build Verifpal. Please review the *Rust Getting Started*[4] instructions in order to understand how to best install Rust for your computer and operating system.

Compiling Verifpal. Once you have installed Git and Rust and cloned the Verifpal repository, all dependencies will be fetched and compiled automatically. Simply type `cargo build --release` to build Verifpal. The binary will then be available under the `target/release/` folder. You may also use `make build`, `make test`, and `make lint` for convenience.

1.6 VERIFPAL FOR VISUAL STUDIO CODE

A Verifpal extension is also available for Visual Studio Code[5].

[3] `https://git-scm.com/book/en/v2/Getting-Started-Installing-Git`

[4] `https://www.rust-lang.org/tools/install`

[5] Visual Studio Code is a free and open source code editor by Microsoft, available for download at `https://code.visualstudio.com/`. Verifpal for Visual Studio Code requires Verifpal 0.40.0 or higher to be installed

Verifpal for Visual Studio Code offers the following features:

- Syntax highlighting for Verifpal models.
- Formatting for Verifpal models using the standard Visual Studio Code API, including support for format-on-save.
- Immediate access to Verifpal documentation and code insights simply by hovering over terms with your cursor.
- Live diagram visualizations of Verifpal models within Visual Studio Code.
- Live analysis of Verifpal models and visual results feedback within Visual Studio Code.

To install Verifpal for Visual Studio Code, simply search for it within the extensions search functionality of your Visual Studio Code Editor.

Syntax highlighting will be available immediately on `.vp` files. To format a model, simply right-click within the editor and select *"Format Document"*.

Hovering over primitives (such as `HKDF` or `AEAD_ENC`) will show documentation for these primitives. Hovering over constants will show their assigned values and the name of the principal that created them. Hovering over queries (such as `confidentiality`) will show a brief description of that query's syntax and functionality.

In order to show a diagram visualizing your protocol, open the Visual Studio Code Command Palette (*Ctrl+Shift+P* on Windows and Linux, *Cmd+Shift+P* on macOS) and search for the *"Verifpal: Show Protocol Diagram"* command.

In order to launch an analysis, open the Visual Studio Code Command Palette (*Ctrl+Shift+P* on Windows and Linux, *Cmd+Shift+P* on macOS) and search for the *"Verifpal: Run Attacker Analysis"* command. It is recommended that this feature not be used for models which take a long time to be analyzed. Using Verifpal in the command line for more complex models will likely yield a better workflow since you will not be able to edit your model while analysis is running.

Verifpal for Visual Studio Code may be configured via the following options in your Visual Studio Code User Settings file:

- `verifpal.enabled`: enables or disables IDE features. (eg. `true`)
- `verifpal.path`: Sets the path for the Verifpal binary on your computer. (eg. `/usr/bin/verifpal`)

1.7 SHARING VERIFPAL MODELS

Verifpal models can be shared and discussed using the Verifpal Workbench, accessible at `https://verifpal.com`. The Workbench runs Verifpal directly in the browser via WebAssembly: you can write, analyze and share models without installing any software.

for all functionality to work correctly.

1.8 NEWS AND DISCUSSION

For the latest news, announcements and discussions regarding Verifpal, please visit `https://verifpal.com`.

CHAPTER 2 THE VERIFPAL LANGUAGE

Now that you've installed Verifpal, you're ready to start describing the protocol you want to verify.

The Verifpal language is the main expressive gateway between you and Verifpal. When describing a protocol in Verifpal, you begin by defining whether the model will be analyzed under a *passive* or *active* attacker. Then, you define the *principals* engaging in activity other than the attacker. These could be Alice and Bob, and perhaps also Charlie. It could be a Server and one or more Clients. It all depends on the protocol that you are describing.

Once you've described the actions of more than one principal, it's time to illustrate the *messages* being sent across the network. Perhaps Alice is initiating a session with Bob, and then sending an encrypted message saying *"hello!"* — or perhaps a TLS connection is being initiated between a Client and a Server, after which a web page is fetched. It's up to you to model these interactions using the Verifpal language.

After having illustrated the principals' actions and their messages, you may finally describe the *queries*, or "questions" that you will ask Verifpal. Can a passive attacker read Alice's first message to Bob? Or perhaps Alice can be impersonated by an active attacker! It's Verifpal's job to help you find out.

2.1 DECLARING THE ATTACKER

First, we must define what kind of attacker Verifpal will use to analyze our model. The syntax for this is pretty simple: `attacker[passive]` indicates a passive attacker, while `attacker[active]` indicates an active attacker.

In Chapter 3, the differences between active and passive attacker are explained in more detail. To summarize, a passive attacker is a malicious *observer* on the network that cannot inject or modify messages. An `active` attacker however can modify messages at will, and inject their own new messages in a bid to obtain as much information and as many different scenarios from the protocol described as it is executed over the network. Their hope is that one of these bits of

information, or that one of these scenarios, will allow them to find a contradiction to the queries posed in the model with regards to the protocol.

2.2 PRINCIPALS

Let's declare a principal `Alice` which knows the public constants `c0, c1` and the private constant `m1`, which will act as the secret message Alice will want to send to Bob later. Since `c0` and `c1` are declared as known publicly, they are immediately also known to the attacker. The same, of course, is not true of `m1`. Alice also *generates* a random value `a`. She will use this value as her private key.

New Principal: Alice

```
^^Iprincipal Alice[
^^I^^I^^Iknows public c0, c1
^^I^^I^^Iknows private m1
^^I^^I^^Igenerates a
^^I^^I]
```

It's that simple! Now, let's proceed with Bob:

New Principal: Bob

```
^^Iprincipal Bob[
^^I^^I^^Iknows public c0, c1
^^I^^I^^Iknows private m2
^^I^^I^^Igenerates b
^^I^^I^^Igb = G^b
^^I^^I]
```

Notice how Bob also calculates `gb = G^b`. Here, `gb` is Bob's public Diffie-Hellman key, while `G^b` quite plainly indicates the standard Diffie-Hellman exponentiation g^b. Later, Alice will be able to write `gb^a`, which is how we illustrate g^{ba} in Verifpal.

2.3 FUNDAMENTAL TYPES IN VERIFPAL

Verifpal has three fundamental types: constants, primitives and equations. A constant may have qualifiers such as *freshness* (if declared using `generates`). Equations are in the form `G^x^y`. Primitives are one of the various built-in functions in Verifpal, and are defined using Verifpal's internal primitive definition structure. All of these elements are touched upon below.

2.3.1 *Constants*

In the above examples, `c0, c1, m1, m2, b, gb` are all *constants*. Certain rules apply on constants in Verifpal:

- *Immutability.* Once assigned, constants cannot be reassigned.
- *Global name-space.* If Bob declares or assigns some constant `c`, Alice cannot declare a constant `c` even if Bob declares or assigns his constant privately.
- *No referencing.* Constants cannot be assigned to other constants, but only to primitives or equations.

These rules exist in order to encourage you to write Verifpal models that will hopefully be cleaner and easier to read.

Let's summarize the different ways that exist to declare constants, and how they differ from one another:

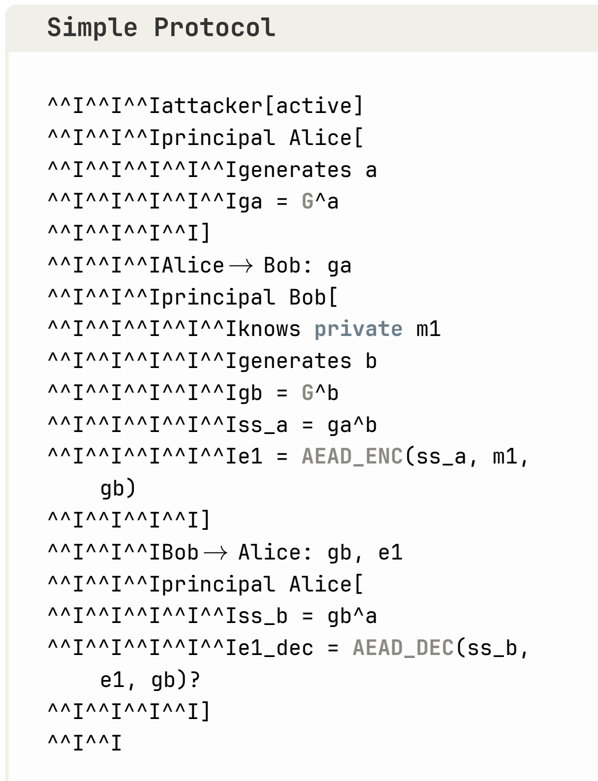
- `knows`: A principal may be described as having prior knowledge of a constant. The qualifiers `private` and `public` describe whether this constant that they have knowledge of is supposed to be considered known by everyone else (including the attacker) or just by them. Constants declared this way are considered to be, well, constant, across every execution of the protocol (i.e. they are not unique for every different time the protocol is executed). A third qualifier, `password`, can be used to declare private constants that are weak or guessable: if they are used directly within, for example, an encryption primitive, and the ciphertext is obtained by the attacker, the attacker will be able to obtain the password value immediately. Therefore, in order to be used safely, values declared using `knows password` must first be sent through a password hashing primitive such as `PW_HASH`.
- `generates`: This allows a principal to describe a *"fresh"* value, i.e. a value that is regenerated every time the protocol is executed. A good example of this could be an ephemeral private key. Such values (and all values derived using these values) are not kept between different protocol session executions.
- `leaks`: This allows us to specify that the principal will leak an existing constant that they already know to the attacker, rendering the value immediately knowable to the attacker at the point of leakage.
- *Assignment*: A constant may be declared by assigning it to the result of a primitive or equation expression. But remember: constants may not be assigned to other constants.

2.3.2 *Primitives*

In order to describe cryptographic protocols, we will of course need cryptographic primitives.

In Verifpal, cryptographic primitives are modeled as ideal abstractions within the Dolev-Yao symbolic model [4]. That is to say, a hash function is an uninterpreted function symbol: the only way to obtain `HASH(x)` is to know `x` and apply `HASH`; properties such as collision resistance or length extension are not modeled. Similarly, the only way to recover a plaintext from `ENC(k, m)` is to possess the key `k`; notions such as key length or block cipher mode are abstracted away[1].

[1] This is the fundamental difference between tools like Verifpal, ProVerif and Tamarin, which operate in the *symbolic* (Dolev-Yao) model, and software like CryptoVerif [5] which operates in the computational model. The computational model allows CryptoVerif to reason about concrete cryptographic properties such as key length, which can make for more accurate modeling in some instances.



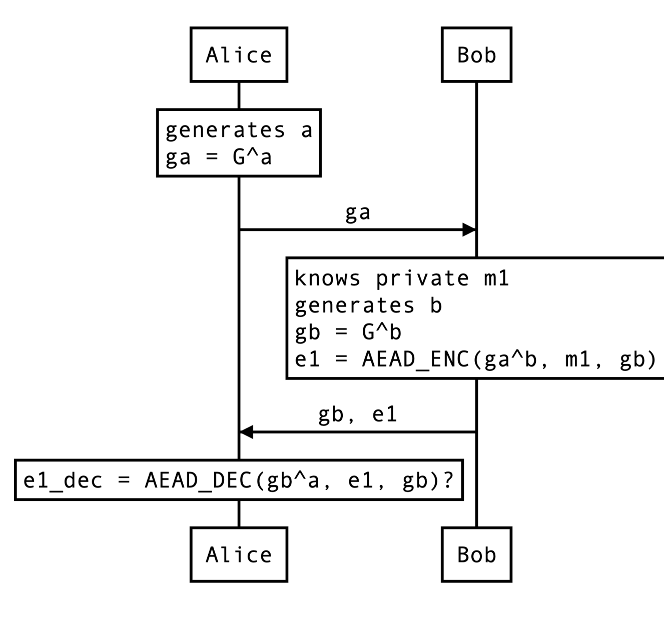


Figure 2.1: A complete example model of a simple protocol is shown on the left. On the right, a helpful diagram is provided to illustrate how modeling in Verifpal works. The diagram on the right is not part of Verifpal's modeling language and is simply provided here as a visual aid.

Internally in Verifpal, every primitive is defined using a common specification structure called PrimitiveSpec. This ensures that all primitives, whether cryptographic or not, are governed by the same set of uniform rules. Aside from the primitive's name, arity (the number of allowed inputs), and the number of allowed outputs, each PrimitiveSpec defines a primitive's behavior solely via a combination of four standard rules:

- Decompose. Given a primitive's output and a defined subset of its inputs, reveal one of its other inputs. For example, given `ENC(k, m)` and key `k`, reveal plaintext `m`. This models the attacker's ability to "open" a ciphertext when they possess the key. Note that Decompose also specifies which arguments (if any) are *passively* revealed: for example, the associated data `ad` in `AEAD_ENC(k, m, ad)` is always visible to any observer, even without the key.

- Recompose. Given a defined subset of a primitive's outputs, reveal one of its inputs. For example, given any two shares `a, b` out of `a, b, c = SHAMIR_SPLIT(x)`, recover the original secret `x`. This models threshold reconstruction in secret sharing.

- Rewrite. Given a matching pattern within a primitive's inputs, rewrite the entire primitive expression to a simpler value. For example, `DEC(k, ENC(k, m))` rewrites to `m`, because decryption with the correct key recovers the plaintext. This is Verifpal's mechanism for modeling the equational theory of cryptographic operations: each rewrite rule captures the algebraic relationship between a primitive and its inverse.

- Rebuild. Given a primitive whose inputs are outputs of the same instance of some other primitive, rewrite the expression to the original input. For example, `SHAMIR_JOIN(a, b)` where `a, b, c = SHAMIR_SPLIT(x)` rewrites to `x`. Rebuild operates during symbolic

evaluation (when resolving a principal's state), whereas RECOMPOSE operates during attacker analysis (when the attacker attempts to derive new values from known outputs).

Core Primitives

Verifpal offers the following *"core"* primitives, which perform basic operations that are not necessarily cryptographic in nature, but still often useful in models:

- `ASSERT(MAC(key, message), MAC(key, message)): unused`.
  Checks the equality of two values, and especially useful for checking MAC equality. Output value is not used; see §2.3.2 below for information on how to validate this check.
- `CONCAT(a, b...): c`.
  Concatenates two or more values[2] into one value. *"Concatenation"* is a word often used in computer science to describe joining multiple strings or values together. For example, the concatenation of the strings `cat` and `dog` would be `catdog`.
- `SPLIT(CONCAT(a, b)): a, b`.
  Splits a concatenation back to its component values. Must contain a `CONCAT` primitive as input; otherwise, Verifpal will output an error.

Hashing Primitives

Verifpal offers the following hashing primitives, which aim to capture classical cryptographic hashing, keyed hashing and hash-based key derivation:

- `HASH(a, b...): x`.
  Secure hash function, similar in practice to, for example, BLAKE2s [11].
  Takes between 1 and 5 inputs and returns one output.
- `MAC(key, message): hash`.
  Keyed hash function. Useful for message authentication and for some other protocol constructions.
- `HKDF(salt, ikm, info): a, b...`.
  Hash-based key derivation function inspired by the Krawczyk HKDF scheme [12].
  Essentially, `HKDF` is used to extract more than one key out a single secret value. `salt` and `info` help contextualize derived keys. Produces between 1 and 5 outputs.
- `PW_HASH(a...): x`.
  Password hashing function, similar in practice to, for example, Scrypt [13] or Argon2 [14]. Hashes passwords and produces output that is suitable for use as a private key, secret key or other sensitive key material. Useful in conjunction with values declared using `knows password a`.

[2] The `CONCAT` primitive can be used to concatenate up to 5 values.

Encryption Primitives

Verifpal offers the following encryption primitives, which aim to capture unauthenticated encryption, and authenticated encryption with associated data:

- `ENC(key, plaintext): ciphertext`.
  Symmetric encryption, similar for example to AES-CBC or to ChaCha20.
- `DEC(key, ENC(key, plaintext)): plaintext`.
  Symmetric decryption.
- `AEAD_ENC(key, plaintext, ad): ciphertext`.
  Authenticated encryption with associated data.
  `ad` represents an additional payload that is not encrypted, but that must be provided exactly in the decryption function for authenticated decryption to succeed. Similar for example to AES-GCM or to ChaCha20-Poly1305.
- `AEAD_DEC(key, AEAD_ENC(key, plaintext, ad), ad): plaintext`.
  Authenticated decryption with associated data.
  See §2.3.2 below for information on how to validate successfully authenticated decryption.
- `PKE_ENC(G^key, plaintext): ciphertext`.
  Public-key encryption.
- `PKE_DEC(key, PKE_ENC(G^key, plaintext)): plaintext`.
  Public-key decryption.

Signature Primitives

Verifpal offers a simple signing primitive with a corresponding signature verification function:

- `SIGN(key, message): signature`.
  Digital signature primitive. Here, `key` is a private key, for example `a`. The corresponding public key is `G^a`.
- `SIGNVERIF(G^key, message, SIGN(key, message)): verified`.
  Verifies whether a signature can be authenticated against the provided public key and message. If key `a` was used for `SIGN`, then `SIGNVERIF` will expect `G^a` as the first argument. Note that the output of a successful `SIGNVERIF` is not the message itself but a verification token; the primary purpose of `SIGNVERIF` is its use as a checked primitive (see §2.3.2 below) to abort the protocol if verification fails.
- `RINGSIGN(key_a, G^key_b, G^key_c, message): signature`.
  Ring signature [15].
  In ring signatures, one of three parties (Alice, Bob and Charlie) signs a message. The resulting signature can be verified using the public keys of all three parties, but the signature does not reveal the signatory — only that they are a member of the signing ring {Alice, Bob, Charlie}. The first argument must be the private key of the actual signer, while the subsequent two arguments must be the public keys of the other two potential signers.

- `RINGSIGNVERIF(G^a, G^b, G^c, m, RINGSIGN(a, G^b, G^c, m)): verified`.
  Verifies whether a ring signature can be authenticated.
  All three public keys of the ring must be provided, but they may be provided in any order and not necessarily in the order used during the `RINGSIGN` operation. As with `SIGNVERIF`, the primary purpose is use as a checked primitive; see §2.3.2 below.

- `BLIND(k, m): blinded`.
  Message blinding primitive, useful for the implementation of blind signatures. Here, the sender uses the secret "blinding factor" `k` in order to blind message `m`, which can then be sent to the signer, who will be able to produce a signature on `m` without knowing `m`. Used in conjunction with `UNBLIND` – see `UNBLIND`'s documentation for more information.

- `UNBLIND(k, m, SIGN(a, BLIND(k, m))): SIGN(a, m)`.
  Once `BLIND(k, m)` is signed by the signer, the sender can convert `SIGN(a, BLIND(k, m))` to `SIGN(a, m)` by unblinding the message using their secret blinding factor `k`. The resulting unblinded signature can then be used as if it were a regular signature by `a` over `m`.

Secret Sharing Primitives

Verifpal offers a simple interface for modeling Shamir Secret Sharing [16], which allows a secret (such as a key) to be split into multiple shares such that only some (and not all) of these shares are required to reconstitute it:

- `SHAMIR_SPLIT(k): s1, s2, s3`.
  In Verifpal, we allow splitting the key into three shares such that only two shares are required to reconstitute it.

- `SHAMIR_JOIN(sa, sb): k`.
  Here, `sa` and `sb` must be two distinct elements out of the set `(s1, s2, s3)` in order to obtain `k`.

Checked Primitives

In Verifpal, `ASSERT`, `SPLIT`, `AEAD_DEC`, `SIGNVERIF` and `RINGSIGNVERIF` are *"checkable"* primitives: if you add a question mark (`?`) after one of these primitives, then that primitive becomes *"checked"*, meaning the protocol execution will abort at that point should the primitive's rewrite rule fail. For example, the session will abort should `AEAD_DEC` fail authenticated decryption (because the key or associated data does not match), should `ASSERT` fail to find its two provided inputs equal, or should `SIGNVERIF` fail to verify the signature against the provided message and public key.

For example: `SIGNVERIF(k, m, s)?` makes this instantiation of `SIGNVERIF` a checked primitive.

Under a passive attacker, Verifpal executes the model once with the honest protocol values. If a checked primitive fails in this single execution, the model is invalid and Verifpal will report an error. Under an active attacker, Verifpal explores many possible executions — each one corresponding to a different combination of attacker mutations (value substitutions) on unguarded

When to Check Primitives

Inevitably, checking every single checkable primitive in your model will lead to fewer attacks on your protocols being found, especially under an active attacker. But is it always accurate to model your protocol this way?

Unchecking certain primitives can make it easier for you to illustrate what could happen if protocol implementations ignore certain real-world *"checks"*, and can lead to some interesting new insights!

wire values. In each such execution, if a checked primitive fails, that particular execution is *truncated* at the point of failure: no subsequent operations for that principal are evaluated. However, values obtained *before* the failure point in that execution are still valid for analysis and may yield new attacker knowledge. The attacker then moves on to explore other mutation combinations. For more information on this, see Chapter 3.

2.3.3 *Equations*

Equations are special expressions intended to capture public key generation (useful for both Diffie-Hellman and signatures), as well as shared secret agreement (useful for Diffie-Hellman).

As we saw earlier, `G^a` indicates the public key obtained from value *a*. This public key can be used both for signing primitives as well as for Diffie-Hellman shared secret agreement. Let's look at some other example equations in Verifpal:

Example Equations

```
^^Iprincipal Server[
^^I^^I^^Igenerates x
^^I^^I^^Igenerates y
^^I^^I^^Igx = G^x
^^I^^I^^Igy = G^y
^^I^^I^^Igxy = gx^y
^^I^^I^^Igyx = gy^x
^^I^^I]
```

In the above, `gxy` and `gyx` are considered equivalent by Verifpal. This reflects the commutativity of Diffie-Hellman exponentiation: $g^{xy} = g^{yx}$. Internally, Verifpal represents equations as chains of exponents rooted at the generator `G`: the equation `gxy = gx^y` is stored as a three-element chain $[\mathtt{G}, x, y]$, and equivalence of such chains is determined by treating the exponents as a commutative multiset (i.e. $[\mathtt{G}, x, y] \equiv [\mathtt{G}, y, x]$).

In Verifpal, all equations must have the constant `G` as their root generator, mirroring Diffie-Hellman behavior. Each individual equation step takes the form `a^b` (a base raised to a single exponent), but equations can be composed by chaining them: `gxy = gx^y` first computes `gx = G^x`, then exponentiates further by `y`, yielding g^{xy}.

2.3.4 *The Equational Theory*

The behavior of Verifpal's primitives can be understood as an *equational theory* — a set of equations that define when two symbolic expressions are considered equal. This is the formal foundation that allows Verifpal to determine, for example, that decrypting a ciphertext with the correct key yields the original plaintext.

The equational theory is defined implicitly by the four rules in each PrimitiveSpec (Decompose, Recompose, Rewrite, Rebuild) and the commutativity of Diffie-Hellman exponentiation. The complete set of equations is:

- $\mathsf{DEC}(k, \mathsf{ENC}(k,m)) \to m$ [rewrite]
- $\mathsf{AEAD_DEC}(k, \mathsf{AEAD_ENC}(k,m,ad), ad) \to m$ [rewrite]
- $\mathsf{PKE_DEC}(sk, \mathsf{PKE_ENC}(g^{sk}, m)) \to m$ [rewrite]
- $\mathsf{SIGNVERIF}(g^{sk}, m, \mathsf{SIGN}(sk,m)) \to nil$ [rewrite]
- $\mathsf{RINGSIGNVERIF}(g^{sk}, \ldots, \mathsf{RINGSIGN}(sk, \ldots)) \to nil$ [rewrite]
- $\mathsf{UNBLIND}(f, \mathsf{SIGN}(sk, \mathsf{BLIND}(f,m)), m) \to \mathsf{SIGN}(sk,m)$ [rewrite]
- $\mathsf{SPLIT}(\mathsf{CONCAT}(a,b,\ldots)) \to (a,b,\ldots)$ [rewrite]
- $\mathsf{ENC}(k,m) \to m$ given k [decompose]
- $\mathsf{AEAD_ENC}(k,m,ad) \to ad$ [passive]
- $\mathsf{BLIND}(f,m) \to m$ given f [decompose]
- $\mathsf{CONCAT}(a,b,\ldots) \to a,b,\ldots$ [passive]
- $\mathsf{SHAMIR_SPLIT}(s)[i] + \mathsf{SHAMIR_SPLIT}(s)[j] \to s$ [recompose: 2-of-3]
- $\mathsf{SHAMIR_JOIN}(s_i, s_j) \to x$ when s_i, s_j are distinct outputs of $\mathsf{SHAMIR_SPLIT}(x)$ [rebuild]
- $g^{ab} = g^{ba}$ [DH commutativity]

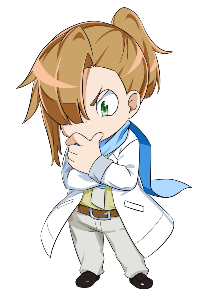
These equations are applied as left-to-right rewrite rules during symbolic evaluation. Verifpal does not reason about equations beyond those listed above: there is no general unification or equational reasoning. This fixed equational theory is what makes analysis decidable and efficient, at the cost of not being able to model user-defined algebraic properties.

2.4 MESSAGES

Sending messages over the network is simple. Only constants may be sent within messages:

Example: Messages

```
^^IAlice → Bob: ga, e1
^^IBob → Alice: [gb], e2
```



Guarding the Right Constants

Verifpal allows you to guard constants against modification by the active attacker. However, guarding all of a principal's public keys, for example, might not reflect real-world attack scenarios, where keys are rarely guarded from being modified as they cross the network.

What interesting new insights will you discover using guarded constants?

Let's look at the two messages above. In the first, Alice is the sender and Bob is the recipient. Notice how Alice is sending Bob her long-term public key `ga = G^a`. An active attacker could intercept `ga` and replace it with a value that they control. But what if we want to model our protocol such that Alice has pre-authenticated[3] Bob's public key `gb = G^b`? This is where *guarded constants* become useful.

In the second message from the above example, we see that, `gb` is surrounded by brackets (`[]`). This makes it a *"guarded"* constant, meaning that while an active attacker can still read it, they cannot tamper with it. In that sense it is *"guarded"* against the active attacker.

2.5 PHASES

Phases allow Verifpal to model temporal security properties such as forward secrecy and post-compromise security. When modeling with an active attacker, a new phase can be declared thus:

Example: Phases

```
^^Iprincipal Alice[...]
^^Iprincipal Bob [...]
^^IBob → Alice: b1

^^Iphase[1]

^^Iprincipal Alice[leaks a2]
```

The semantics of phases are governed by the following rules:

- *Knowledge.* A value is learned by the attacker at the earliest phase in which it is communicated. In the above example, the attacker learns `b1` in phase 0 and `a2` in phase 1.
- *Manipulation.* The attacker can only manipulate (i.e. substitute or inject) a value within phases in which that value is communicated. A value may be communicated in more than one phase (e.g. if Alice sends `a2` to Carol in phase 1 and to Damian in phase 2). Thus, the attacker can manipulate `b1` in phase 0 but not in phase 1, even though the attacker still *knows* `b1` in phase 1.

[3] *"Pre-authentication"* refers to Alice confirming the value of Bob's public key before the protocol session begins. This helps avoid having an active attacker trick Alice to use a fake public key for Bob. This fake public key could instead be the attacker's own public key. We call this a Mayor-in-the-Middle attack.

- *No cross-phase mutation reuse.* Values derived from mutations of `b1` in phase 0 cannot be used to construct new values in phase 1.

Phases are useful to model scenarios where, for example, the attacker manages to steal Alice's keys strictly *after* a protocol has been executed, allowing the attacker to use their knowledge of that key material, but only outside of actually injecting it into a running protocol session.

2.6 QUERIES

A Verifpal model is always concluded with a *queries* block, which contains essentially the questions that we will ask Verifpal to answer for us as a result of the model's analysis. Queries have an important role to play in a Verifpal model's constitution. The Verifpal language makes them very simple to describe, but you may benefit from learning more on how to properly use them in your models. For more information on queries, see Chapter 3. §2.8 below shows a quick example of how to illustrate queries in your model.

2.7 COMMENTS

At any point in your model, you may insert comment lines by prepending them with a double backslash (`//`). Comments are useful to include notes for yourself or others reading your model.

2.8 A SIMPLE COMPLETE EXAMPLE

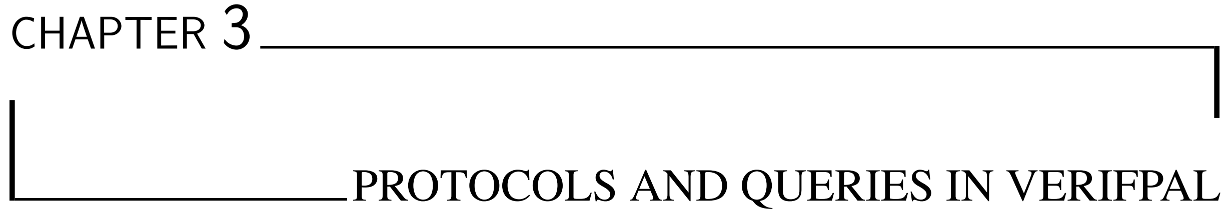
Figure 2.1 provides a full model of a naïve protocol where Alice and Bob only ever exchange unauthenticated public keys (`G^a` and `G^b`). Bob then proceeds to send an encrypted message to Alice using the derived Diffie-Hellman shared secret to encrypt the message. We then want to ask Verifpal the following questions:

1. Can the attacker obtain the ciphertext?
2. Can the attacker obtain the plaintext?
3. Can the attacker impersonate Bob and deliver a tampered ciphertext to Alice that nevertheless still authenticates?
4. Are the shared secrets that are derived between Alice and Bob always equivalent?

Example: Queries

```
^^Iqueries[
^^I^^I^^Iconfidentiality? e1
^^I^^I^^Iconfidentiality? m1
^^I^^I^^Iauthentication? Bob → Alice: e1
^^I^^I^^Iequivalence? ss_a, ss_b
^^I^^I]
```

Under a passive attacker: the attacker can observe `e1` (since it crosses the network), but cannot obtain the plaintext `m1`; no authentication or equivalence contradictions are found. Under an active attacker, all four queries are contradicted: the attacker can obtain both `e1` and `m1`, can impersonate Bob, and can cause the derived shared secrets to differ. Can you figure out why? If not, no need to worry: in Chapter 3, we will learn more about how the Verifpal attacker behaves when analyzing a model. In Chapter 4, we will cover common considerations protocol designers face when building a protocol for a particular use case.



CHAPTER 3

PROTOCOLS AND QUERIES IN VERIFPAL

So far, this manual has assumed that you have an understanding of the kind of thinking that determines the design of a cryptographic protocol: the use cases, the security goals, the principals involved. In this chapter, we will go through these concepts again, as they are central to a complete understanding of Verifpal.

Protocol designers are skilled craftspeople. When Trevor Perrin and Moxie Marlinspike looked at secure messaging protocols, they decided that none of them were good enough: if Alice and Bob were communicating over WhatsApp, they deserved that their messages would remain safe across the wire even if Alice's phone were to be stolen. In creating the Signal protocol, they achieved the highest level of security publicly available for secure messaging, and by making their protocol design open and efficient, ensured that it would be implemented across billions of devices.

Similarly, when Jason Donenfeld looked at existing VPN solutions, he found protocols that had to deal with decades of outdated cryptography, dozens of different versions and configurations (many of them insecure) spread across tens of thousands of lines of code. In designing WireGuard, he was able to capture security goals more ambitious and advanced than those captured by any other mainstream VPN solution, and in code that was a fraction of the size.

When Eric Rescorla led the TLS 1.3 effort, he was able to conduct a worldwide community towards agreeing on a new standard for encrypting the majority of web communications, doing so in a way that eliminated attacks discovered by tools similar to Verifpal and making the protocol itself simpler at the same time.

All a protocol designer has to do is capture an elegant construction and illustrate it once. If it is shown to satisfy their chosen security goals, then it can immediately become a benefit to the privacy and safety of billions of people across the world. Wouldn't it be amazing if you could learn how to think like these pioneers? Let's take a look at how we can use Verifpal to prototype our own protocols.

Are You Sure It's Private?

It might be tempting to mark a declared value as private; but is it really? You could declare a key embedded into a smartphone as `knows private builtInKey`, but it might not be so private if that smartphone's hardware is reverse engineered. Be careful when declaring values as private, and only do so when you're sure they will be. Otherwise you might model your protocol as stronger than it truly is.

3.1 USE CASES AND SECURITY GOALS

Naturally, the first thing you want to consider when designing a protocol is the *use case*. Will it be a protocol for encrypting and authenticating phone calls, such as DTLS-SRTP? Will it be a protocol for encrypted video chat, such as WebRTC? Or maybe you're looking to test out your new protocol for communications between an ATM and its host bank. In a world where even refrigerators and toasters are connecting to the Internet, there certainly is no shortage of use cases to consider.

Once you've determined your use case, you will need to determine the *principals* and the *security goals* that they are supposed to benefit from by engaging in your protocol. *"Principals"* is just a fancy word for *"parties involved in your protocol."* In a secure messenger, that's Alice and Bob. In a secure *group* chat, however, that could go from Alice and Bob all the way to Yvonne and Zachary. In HTTPS, you've got the old client (your browser) and server (the website you're connecting to.) And so it goes.

Once you've identified your principals, it's important for you to be very clear about what your expectations are with regards to their security goals. Sure, you could expect communications between client and server to be *confidential* against an active attacker, but would that hold if the server were to be impersonated by an attacker? If you're hoping for that to be true, then you better check for message *authentication* as well.

It is possible that the protocol you are modeling has sessions that could go in an arbitrary number of directions. Take for example group secure messaging protocols, where Alice, Bob, Charlie and Danielle are communicating in an end-to-end encrypted group. Will you model Alice as sending a message, with Bob then replying? Will you model Danielle sending three messages in a row without anyone responding? How does Danielle doing so affect the forward secrecy guarantees of these messages? Are they as secure as the messages Danielle could have sent, had she waited to first receive a reply from someone else in the group (which could also contain fresh key material)? What about Evan, a fifth participant, who joins the group chat halfway through. Is he able to read communications sent before he joined? Is that a desired property of the protocol?

In Verifpal models, you will be constrained to modeling one protocol execution scenario: in such circumstances, it might be worthwhile to have different models for the same protocol, illustrating situations where different events occur in a different order. By applying the same queries across different models covering different scenarios, you can better understand how your protocol holds up in different circumstances.

```
Example Queries

^^I^^I^^Iqueries[
^^I^^I^^I^^I^^Iconfidentiality? m1
^^I^^I^^I^^I^^Iauthentication? Alice → Bob: e1
^^I^^I^^I^^I]
^^I^^I
```

Figure 3.1: Example confidentiality and authentication queries.

3.2 QUERIES

In Chapter 2, we saw how the Verifpal language allows us to describe protocols simply and clearly. Once we've written our protocol down, however, analysis must begin: it's time to ask Verifpal the hard questions we want answered about the security of our design.

By defining queries, we will be able to formulate the questions we have regarding our protocol so that Verifpal can understand them. Then, by reading the output of the analysis under an active or a passive attacker, we can learn more about the properties and limitations of the protocol that we have described. Does your protocol really protect the confidentiality of messages from an active attacker? In what situations does it allow a malicious interceptor to impersonate one of the parties? Queries are how we ask Verifpal these questions, and the goal of protocol analysis is to obtain useful and insightful answers.

In Figure 3.1, we see two different types of queries. Let's go in depth into what each of them means and how we can use them to test for different properties.

3.2.1 *Confidentiality Queries*

Confidentiality queries are the most basic of all Verifpal queries. In the example confidentiality query shown in Figure 3.1, we ask: *"can the attacker obtain `m1`?"* — where `m1` is a sensitive message. Formally, a confidentiality query for value *v* is *contradicted* if and only if the attacker can derive a value equivalent to the resolved form of *v* through any combination of knowledge gathering, decomposition, reconstruction, and (under an active attacker) message mutation. If the query is contradicted, the attacker was able to obtain the value despite it being presumably protected.

A passive attacker would have to rely on the encryption key for `m1`'s ciphertext `e1` being somehow communicated on the network, whether explicitly or in terms of its components, in order to obtain `m1`. An active attacker, however, could have replaced Bob's public keys as they were sent to Alice, before Alice could use them to encrypt `m1`. Read on to §3.3 to learn more.

3.2.2 *Authentication Queries*

Authentication queries are a bit trickier than confidentiality queries. In the example authentication query shown in Figure 3.1, we ask: *"if Bob successfully uses `e1` in his protocol operations, does that necessarily mean that Alice sent `e1` to Bob?"* The implication is that if the attacker

was able to successfully convince Bob to accept and use `e1`, then an impersonation attack could have occurred where the attacker was able to impersonate Alice.

Authentication queries rely heavily on Verifpal's notion of *"checked"* or *"checkable"* primitives, as defined in §2.3.2.

Formally, for an authentication query `authentication? Alice → Bob: e1`, Verifpal checks whether the constant `e1` is used in any primitive within Bob's (the recipient's) state, and if so, whether the value's *sender* in that slot matches Alice (the claimed sender). The sender is tracked through Verifpal's provenance system: each constant carries metadata recording who created it, who sent it, and whether the attacker tampered with it. If there exists a protocol execution in which Bob uses `e1` in a primitive but the sender recorded in that slot is not Alice — for example, because an active attacker intercepted and replaced `e1` — then the authentication query is contradicted.

In terms of Lowe's hierarchy of authentication specifications [17], Verifpal's authentication queries most closely correspond to a notion of *non-injective agreement*: if Bob completes a protocol run apparently with Alice, then Alice has indeed sent the relevant values, though the query does not by itself check that each run of Bob's corresponds to a unique run of Alice's (i.e. it does not check for *injective* agreement, which would additionally guard against replay attacks).

Note that we don't check for the authentication of plaintext `m1` — that is because `m1` is only obtainable by Bob once decryption succeeds, which only happens if `AEAD_DEC` is successfully rewritable back into the input values to `AEAD_ENC`, i.e. if the primitive passes the check.

In Figure 3.4, we see authentication queries applied not only to messages exchanged between Alice and Bob, but also to Bob's *"signed pre-key"*[1].

3.2.3 *Freshness Queries*

Freshness queries are useful for detecting replay attacks, where an attacker could reuse a value from one session in another session. A value is considered *fresh* if at least one of its transitive components was declared via `generates` (i.e. is randomly generated per session and is not static across sessions) and has not been leaked. Since generated values are assumed to be unique to each session, any value derived from them will differ between sessions, preventing replay.

Formally, for a freshness query `freshness? v`, Verifpal resolves `v` in the principal's state and recursively checks whether the resolved value contains at least one constant that was declared via `generates` and that has not been leaked. If it does, the value is fresh and the query passes. If it does not — for example, because the value is derived entirely from static (`knows`) constants, or because an active attacker replaced all fresh components with known, non-fresh values — then the query is contradicted. In Figure 3.2, the first freshness query will be contradicted (because `ha = HASH(a)` derives only from the static private key `a`), while the second will not (because `hb = HASH(b)` derives from the generated value `b`).

[1] For more information on what a *"signed pre-key"* is and how the Signal protocol works, see Chapter 5.

```
Example Freshness Query

^^I^^I^^Iattacker[active]
^^I^^I^^Iprincipal Alice[
^^I^^I^^I^^I^^Iknows private a
^^I^^I^^I^^I^^Igenerates b
^^I^^I^^I^^I^^Iha = HASH(a)
^^I^^I^^I^^I^^Ihb = HASH(b)
^^I^^I^^I^^I]
^^I^^I^^IAlice → Bob: ha, hb
^^I^^I^^Iprincipal Bob[
^^I^^I^^I^^I^^Iknows private a
^^I^^I^^I^^I^^I_ = ASSERT(ha, HASH(a))
^^I^^I^^I^^I]
^^I^^I^^Iqueries[
^^I^^I^^I^^I^^Ifreshness? ha
^^I^^I^^I^^I^^Ifreshness? hb
^^I^^I^^I^^I]
^^I^^I
```

Figure 3.2: Example freshness queries.

3.2.4 *Unlinkability Queries*

Protocols such as DP-3T (see Chapter 7), voting protocols and RFID-based protocols posit an "unlinkability" security property on some of their components or processes. While formal definitions for unlinkability exist in the literature (e.g. [18]), they vary in scope and formulation. In Verifpal, we adopt the following working definition: *"for two observed values, the adversary cannot distinguish between a protocol execution in which they belong to the same user and a protocol execution in which they belong to two different users."*

Formally, for an unlinkability query evaluating two or more values (e.g. `unlinkability? a, b, c`), Verifpal performs two checks:

1. **Freshness check.** Verifpal first verifies that every queried constant satisfies freshness (i.e. each transitively contains at least one non-leaked generated value). If any constant fails this check, the query is immediately contradicted: a value that is static across sessions can trivially be used to link executions. This mirrors the freshness query semantics.
2. **Common-source check.** If all constants are fresh, Verifpal then checks whether any two of the queried constants resolve to *equivalent* values (same primitive with same arguments, differing only in output index). If such a pair is found, Verifpal further checks whether the attacker can *reconstruct or recompose* the underlying primitive from known values. If so, the query is contradicted: the attacker can determine that the two values are outputs of the same computation and thereby link them. For example, if `h1, h2, h3 = HKDF(a, b, nil)` and the attacker knows `a` and `b`, then the attacker can reconstruct the `HKDF` and determine that `h1` and `h2` share a common source.

This two-part analysis may not capture all notions of unlinkability in the literature, and future

```
Example Unlinkability Query

^^I^^I^^Iattacker[active]
^^I^^I^^Iprincipal Alice[
^^I^^I^^I^^I^^Igenerates b
^^I^^I^^I^^I]
^^I^^I^^IAlice → Bob: b
^^I^^I^^Iprincipal Bob[
^^I^^I^^I^^I^^Iknows private a
^^I^^I^^I^^I^^Igenerates c
^^I^^I^^I^^I^^Igenerates d
^^I^^I^^I^^I^^Ileaks c
^^I^^I^^I^^I^^Ih1, h2, h3 = HKDF(a, b, nil)
^^I^^I^^I^^I^^Ih4, h5, h6 = HKDF(c, c, nil)
^^I^^I^^I^^I^^Ih7, h8, h9 = HKDF(a, c, d)
^^I^^I^^I^^I]
^^I^^I^^Iqueries[
^^I^^I^^I^^I^^Iunlinkability? h1, h2, h3
^^I^^I^^I^^I^^Iunlinkability? h4, h5, h6
^^I^^I^^I^^I^^Iunlinkability? h7, h8, h9
^^I^^I^^I^^I]
^^I^^I
```

Figure 3.3: Example unlinkability queries.

versions of Verifpal may expand this definition. Figure 3.3 shows an example of a model with three unlinkability queries. Verifpal will only be able to contradict the first two queries.

3.2.5 *Equivalence Queries*

For many protocols, it is useful to check whether shared secrets derived independently by Alice and Bob are equivalent in all completed executions of the protocol. This is especially relevant for Diffie-Hellman key exchanges, where Alice computes g^{ba} and Bob computes g^{ab}: these must be equivalent for the protocol to function correctly, but an active attacker who substitutes public keys during transit could cause them to diverge.

Formally, for an equivalence query `equivalence? ss_a, ss_b`, Verifpal resolves both constants in the current principal's state and checks whether their resolved values are *structurally equivalent* — that is, whether they represent the same symbolic term under Verifpal's equational theory (including Diffie-Hellman commutativity). If the query passes, then in every execution that Verifpal explores, `ss_a` and `ss_b` resolve to the same underlying value. A contradiction means the attacker found a way — for example, via substitution of an unguarded public key — to cause the two values to diverge. §2.8 shows an example of an equivalence query in action.

3.2.6 *Advanced Security Goals*

In addition to confidentiality and authentication, Verifpal is able to model for an advanced security goal known as *key compromise impersonation*.

Many protocols, including Signal and WireGuard, assume that if Alice's long-term keys are

Figure 3.4: Verifpal results for a model of the Signal protocol. Here, we did not bother to guard Alice or Bob's long-term keys. Therefore, despite a correct execution of the protocol and despite *"checking"* all signature verifications, the attacker was able to find contradictions to all queries.

compromised, then the attacker may impersonate her to others. This is a natural and expected assumption: the goal of long-term keys is to provide a sense of permanent identity to their owner. However, in protocols suffering from a *key compromise impersonation* vulnerability, compromising Bob's long-term keys also allows the attacker to impersonate Alice *to* Bob. One such protocol is Signal, and you can learn more about how key compromise impersonation is modeled using Verifpal in Chapter 5.

And what about ephemeral keys? In the protocols we've considered and cited so far, the goal of ephemeral keys is to provide security properties known as *forward secrecy* and *post-compromise security* [7]. The former asks the question: *"does stealing Alice's device allow the thief to decrypt messages she sent in the past?"*, while the latter asks whether the protocol can *recover* its security guarantees after a compromise — that is, whether messages sent sufficiently long after a key compromise are once again protected.

Verifpal currently supports basic forward secrecy checks using phases, which are discussed in §2.5. Chapter 5 shows an analysis of forward secrecy properties in the Signal secure messaging protocol.

3.2.7 *Query Options*

Imagine that we want to check, in the following model, if Alice will only send some message to Carol if it has first authenticated it from Bob:

Query Options Example

```
^^Iattacker[active]
^^Iprincipal Bob[
^^I^^I^^Iknows private psk
^^I^^I^^Igenerates m
^^I^^I^^Ie = ENC(psk, m)
^^I^^I^^Ih = MAC(psk, e)
^^I^^I]
^^IBob → Alice: e, h
^^Iprincipal Alice[
^^I^^I^^Iknows private psk
^^I^^I^^I_ = ASSERT(MAC(psk, e), h)?
^^I^^I^^Im2 = DEC(psk, e)
^^I^^I]
^^IAlice → Carol: [m2]
^^Iprincipal Carol[
^^I^^I^^I_ = HASH(m2)
^^I^^I]
```

This can be accomplished by adding the `precondition` option to the authentication query for `e`:

Query Options Example (Cont.)

```
^^Iqueries[
^^I^^I^^Iauthentication? Bob→ Alice: e[
^^I^^I^^I^^I^^Iprecondition[Alice→ Carol: m2]
^^I^^I^^I^^I]
^^I^^I]
```

The above query essentially expresses: *"The event of Carol receiving `m2` from Alice shall only occur if Alice has previously received and authenticated an encryption of `m2` as coming from Bob."*

This syntax allows us to obtain some insight on the communication of `m2` based on the result of other queries, and to also link the authentication of that communication on the authentication of other values, which can be important when `m2` is being communicated as a guarded constant, which is the case in the above.

Right now, `precondition` is the only available kind of query option, but other kinds of query options may be added in future releases of Verifpal.

3.3 PASSIVE AND ACTIVE ATTACKERS

Verifpal's goal is to obtain as many values as it is logically possible from the viewpoint of an attacker on the network, and to check whether any of those values or the scenarios in which they are used contradict the queries posed in the model.

As a *passive* attacker, Verifpal observes all values exchanged between principals (i.e. all wire traffic) and applies a fixed-point deduction loop: it repeatedly attempts to *decompose* known values (e.g. decrypt a ciphertext using a known key), *reconstruct* new values from known components, and identify *equivalent* values, until no new knowledge is gained. The attacker cannot modify any messages.

As an *active* attacker, Verifpal additionally has the power to *mutate* unguarded values as they cross the network — that is, to replace any unguarded constant in any message with a value of its choosing, including values it has crafted. Each different combination of mutations corresponds to a different protocol execution. Verifpal systematically explores these executions via a bounded-depth search (see Chapter 4), running the deduction loop to its fixed point and then checking queries as a separate phase after each execution. Fresh values (declared via `generates`) are assumed to be different in every session, so they and all values transitively derived from them are not carried across different protocol executions.

The active attacker can also generate its own values, such as a key pair that it controls (e.g. `G^nil`), and use these as substitutes for any unguarded values sent between principals. If, during a protocol execution, a checked primitive fails, that particular execution is *truncated* at the point of failure: no subsequent operations for that principal are evaluated. However, all values obtained before the failure are retained in the attacker's knowledge and may be useful in subsequent executions.

Verifpal tracks the *provenance* of every value: who first declared or generated it (*creator*), who sent it (*sender*), and whether the attacker tampered with it (*attacker_tainted*). This metadata is used to analyze authentication queries — in particular, to determine whether a value used by a principal in a checked primitive was actually sent by the claimed sender.

While analysis under a passive attacker may seem restricted, it is sometimes useful to be able to consider this weaker attacker model in order to model for circumstances and use cases where we do not expect our system to ever be under active attack. For example, an air-gapped[2] control center for a nuclear power plant could be reasonably analyzed under a passive attacker, since all principals could be assumed to have obtained some high-level security clearance.

Let's review the more serious capabilities granted to an active attacker:

Modifying values within messages. An active attacker can replace `e1` with `e2` or anything else that it chooses as that value is being sent in a message from Alice to Bob. While that would result in Bob receiving the modified value, note that Alice's state would still indicate her possession of an intact `e1`, since an active attacker cannot influence the local state of any principal. Note that, as described in §2.4, an active attacker is unable to modify any guarded constants as they are sent within messages, despite being able to read them.

Crafting and injecting malicious values. An active attacker can also choose to replace Alice's public key `G^a` with their own crafted public key `G^attacker`, where the attacker has generated and controls `attacker`. In many protocols, including the one described earlier in §2.8, this can have disastrous consequences.

Executing an unbounded number of sessions. An active attacker can run the protocol an unbounded number of times. Not only that, but the attacker can also keep information learned in previous protocol executions and re-use it in future executions. There is one exception to this: if a learned value is composed of at least one *generated* value (declared using `generate`, see §2.3.1), then it cannot be kept across protocol executions, since that component is assumed to be randomly and freshly generated each session.

Active attacker analysis is more likely to resemble the threat model of the protocol you are analyzing: it applies to any reasonable analysis of HTTPS, secure messaging, VPN, SSH communication and much more.

When analyzing under an active attacker, guarded constants and checked primitives become much more important to employ correctly. For example, you may want to make sure that when Alice and Bob exchange long-term public keys, these values are guarded against modification against an active attacker. This is how we can model *mutual authentication* in Verifpal. You may also want to check certain signature verification (`SIGNVERIF`) or authenticated decryption (`AEAD_DEC`) operations such that the protocol aborts if they fail. Chapter 5 talks more about these scenarios in detail, since they are salient to our analysis of Signal in Verifpal.

[2] *"Air-gapped"* is a term used to describe a system that is cut off or isolated from any other system or network. For example, a computer network can be considered air-gapped if it is only accessible via a single physical keyboard, not connected to the Internet, etc.

Results and Scenarios

Suppose for example that you model an authentication query for a message that Alice sends to Bob, and which Bob never reads. No contradictions are found — this surprises you! Does it mean the message was authenticated despite Bob not reading it? No! What Verifpal is trying to say is that no scenario was found in which Bob reads an unauthenticated message. Remember: queries without contradictions mean that no contradicting *scenarios* were found.

3.4 UNDERSTANDING VERIFICATION RESULTS

Figure 3.4 gives us the results of Verifpal's analysis of Signal, with no mutual authentication of Alice and Bob's long-term public keys, and with only `SIGNVERIF` as a checked primitive. Let's try to understand what the results shown in Figure 3.4 mean for each query.

- `confidentiality? m1`: An active attacker was able to decrypt `m1` since they can impersonate both Alice and Bob due to their not authenticating their long-term public keys[3] (or expressing that authentication using guarded constants).
- `authentication? Bob → Alice: gbs`: Here, Verifpal is telling us that Bob's signed prekey could have been signed by an active attacker instead using a signing private key that they control. The active attacker could then substitute Bob's long-term signing public key with their own as it is being sent to Alice, leading Alice to successfully verify the signature under the malicious public key.
- `authentication? Alice → Bob: e1`: Since the active attacker is able to decrypt `e1` as well as fully impersonate Alice to Bob due to a full mayor-in-the-middle attack, then the attacker could have sent their own `m1` or replacement message value, thereby making it appear as if this message was sent by Alice whereas that is not necessarily the case.
- `confidentiality? m2`: Similarly to `m1`, an active attacker was able to decrypt `m2` due to their ability to fully impersonate both parties.
- `authentication? Bob → Alice: e2`: Since the active attacker is able to decrypt `e2` as well as fully impersonate Bob to Alice due to a full mayor-in-the-middle attack, then the attacker could have sent their own `m2` or replacement message value, thereby making it appear as if this message was sent by Bob whereas that is not necessarily the case.

Had we guarded Alice and Bob's long-term public keys in our model, the results of this analysis would have been markedly different; we will look into this in detail in Chapter 5.

3.5 MODELING A CHALLENGE-RESPONSE PROTOCOL

Figure 3.5 shows a simple challenge-response protocol written in Verifpal. While it is demonstrated here as a complete protocol, challenge-response mechanisms are a common component

[3] `(a3dh, asig)` represents Alice's long-term Diffie-Hellman and signing private keys, while `(b3dh, bsig)` represents Bob's.

Challenge-Response Protocol

```
^^I^^Iattacker[active]
^^I^^Iprincipal Server [
^^I^^I^^I^^Iknows private s
^^I^^I^^I^^Igs = G^s
^^I^^I^^I]
^^I^^Iprincipal Client[
^^I^^I^^I^^Iknows private c
^^I^^I^^I^^Igc = G^c
^^I^^I^^I^^Igenerates nonce
^^I^^I^^I]
^^I^^IClient → Server: nonce
^^I^^Iprincipal Server[
^^I^^I^^I^^Iproof = SIGN(s, nonce)
^^I^^I^^I]
^^I^^IServer → Client: gs, proof
^^I^^Iprincipal Client[
^^I^^I^^I^^Ivalid = SIGNVERIF(gs, nonce, proof)
^^I^^I^^I^^Igenerates attestation
^^I^^I^^I^^Isigned = SIGN(c, attestation)
^^I^^I^^I]
^^I^^IClient → Server: [gc], attestation, signed
^^I^^Iprincipal Server[
^^I^^I^^I^^Istorage = SIGNVERIF(gc, attestation, signed)?
^^I^^I^^I]
^^I^^Iqueries[
^^I^^I^^I^^Iauthentication? Server → Client: proof
^^I^^I^^I^^Iauthentication? Client → Server: signed
^^I^^I^^I]
^^I
```

Figure 3.5: A simple challenge-response protocol in Verifpal.

of many larger protocols. The goal here is to for Client to challenge Server to prove ownership of a signing key pair `(s, gs = G^s)`. Client decides to do this by generating a random `nonce`[4] that it then sends to Server. The challenge is for Server to produce a valid signature for that nonce using `s`, thereby proving that they own `gs`. Since the Server cannot choose or predict `nonce`, they are forced to use the value provided by Client.

Does Figure 3.5 correctly capture this challenge-response mechanism? The answer is *no*: there are two missing elements to this model before it is correct. Can you determine what they are?

First, if we analyze this protocol as it is described in Verifpal, then Client will send `valid` to the server whether or not `SIGNVERIF` succeeds. Therefore, we must *check*[5] `SIGNVERIF` by adding a `?` at the end of that line. Now, Client will not send `valid` unless signature verification passes.

Second, nothing is preventing an active attacker from conducting a mayor-in-the-middle attack and replacing `gs = G^s` with `gs = G^a_0`, where `a_0` is a private signing key controlled by the attacker. Therefore, we can conclude that this challenge-response protocol is only secure against an active attacker if `gs` is *guarded* as it is transmitted from Server to Client. Marking `gs` as a guarded constant[6] makes it impossible for the value to be replaced by an active attacker. Practically, it implies that Client has pre-authenticated Server's signing public key.

Such considerations help illustrate the sort of thing you'll need to watch out for when designing, modeling and analyzing protocols. In Part II of this manual, we will look at how tweaking existing models, once they are written, allows us to quickly prototype our protocol in slightly different scenarios and to see whether they same security goals are achieved.

[4] *"Nonce"* is a common term used in cryptography to indicate a randomly chosen value that is never used more than once — i.e. a **n**umber used **once**.

[5] See §2.3.2 for more information on checked primitives.

[6] See §2.4 for more information on guarded constants.

CHAPTER 4 ANALYSIS IN VERIFPAL

Verifpal is a protocol analysis tool; unlike some other automated formal verification tools [5], it does not produce game-based proofs of the protocols that it analyzes. Instead, it digests models representing the execution of a protocol under a specific scenario enacted by principals that act in a specific way. Verifpal's goal is to attempt to find contradictions to the queries presented by the user. In order to do this, it follows a specific, formalized analysis methodology.

4.1 THE VERIFICATION PIPELINE

Before analysis begins, Verifpal processes the model through a multi-step pipeline:

1. **Parsing.** The `.vp` model file is parsed into an abstract syntax tree (AST) representing the attacker type, principals with their expressions, messages, phases, and queries.
2. **Sanity checking.** The AST is validated for structural correctness: phases must be sequential, all principals referenced in messages and queries must be declared, primitive arities must match their specifications, checked primitives must be applied only to checkable primitives, and Diffie-Hellman equations must use `G` as their root generator.
3. **Construction.** The validated model is transformed into two internal representations:
   - A *protocol trace* — a global, immutable record of every constant's initial value, its creator, the principals that know it, and the phases in which it is communicated.
   - A set of *principal states* — per-principal snapshots recording each constant's resolved value along with metadata about whether it was received over the wire, whether it is guarded, and which principals are capable of mutating it.
4. **Verification.** The analysis engine runs, branching on attacker type (passive or active), as detailed in the following sections.

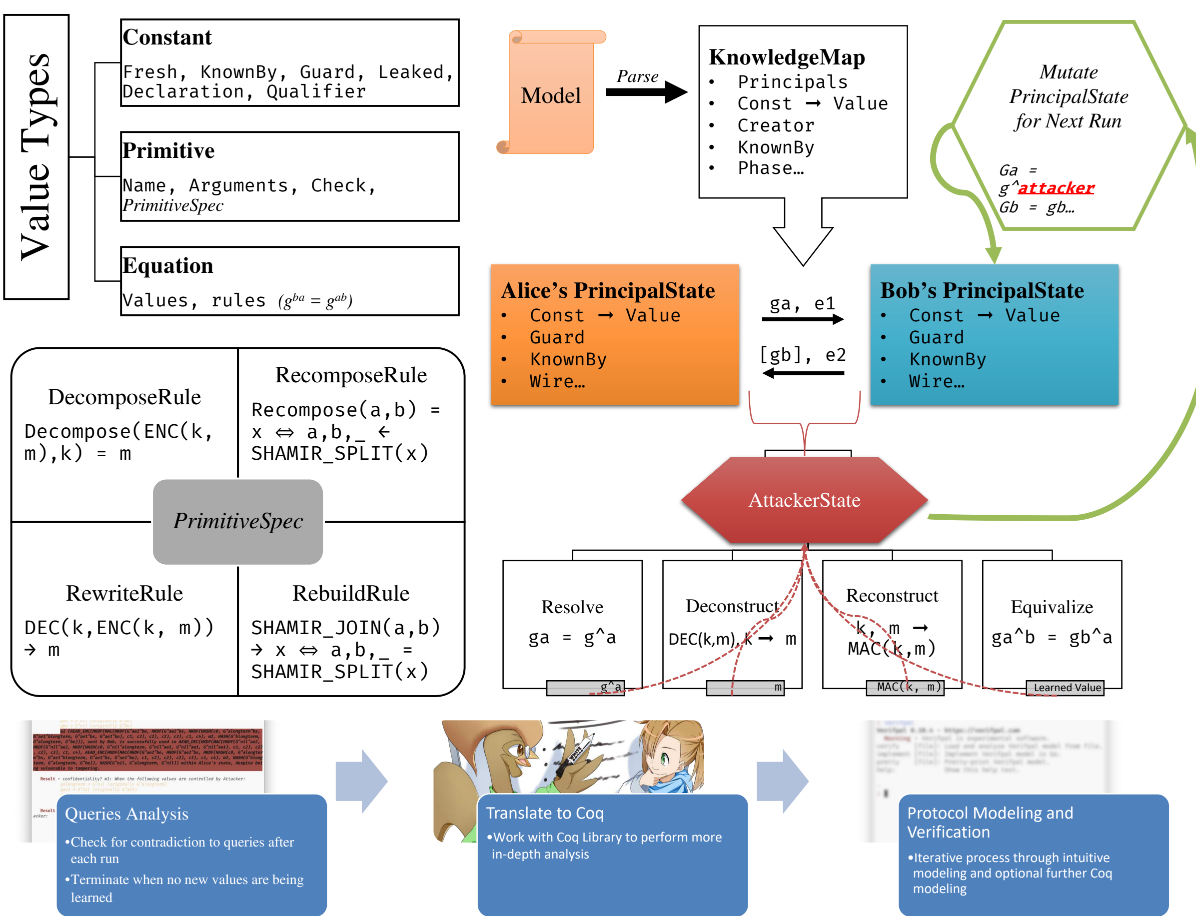


Figure 4.1: Verifpal analysis methodology. On the left, the three fundamental types usable in Verifpal models are illustrated. As noted in §2.3, all cryptographic primitives are defined via a standard PrimitiveSpec structure, which adapts a primitive's definition via a combination of four rules. On the right, a model analysis is illustrated: first, the Verifpal model is parsed and translated into a global immutable protocol trace from which a principal state is derived for each declared principal. Based on the messages exchanged between these principal states, the attacker obtains values to which it can apply the deduction rules discussed. The attacker keeps doing this until it reaches a fixed point, at which point (under an active attacker) it mutates the model and re-runs analysis.

4.2 Analysis Methodology

Verifpal's analysis methodology (Figure 4.1) consists of two components: a *deductive analysis loop* that derives all attacker-obtainable values from a given protocol execution, and (for active attackers) a *mutation search* that systematically explores different protocol executions by substituting values on the wire. For each mutation combination, Verifpal runs a strict three-phase pipeline: *trace generation* (resolve values, apply rewrites, inject skeletons), *knowledge closure* (run the deduction loop to its fixed point), and *query evaluation* (check all queries against the final attacker knowledge). These phases are never interleaved — the deduction loop runs to completion before any queries are checked.

4.2.1 *The Deductive Analysis Loop*

The core of Verifpal's analysis is a fixed-point deduction loop. Given the current attacker knowledge $\mathcal{V}_A$ and a principal's state $\mathcal{V}_P$, the loop repeatedly applies three phases of deduction rules until no new values can be derived:

1. **Phase 1: Decompose.** For each value in $\mathcal{V}_A$, attempt two operations:
   - *Active decomposition*: if the value is a primitive and the attacker knows the required inputs (e.g. the decryption key), extract the hidden argument according to the primitive's DECOMPOSE rule.
   - *Passive decomposition*: extract any arguments that the primitive's specification marks as passively revealed.
2. **Phase 2: Reconstruct and Recompose.** For each value in $\mathcal{V}_P$ (the principal's assigned values):
   - *Reconstruct*: check whether the attacker possesses all the component values needed to build this value from scratch. If so, add it to $\mathcal{V}_A$. This is applied recursively with a bounded depth to handle nested primitives and equations (e.g. constructing a Diffie-Hellman shared secret from known exponents).
   - *Recompose*: check whether the attacker possesses a sufficient subset of a multi-output primitive's outputs to recover one of its inputs, according to the primitive's RECOMPOSE rule (e.g. recovering a secret from two out of three Shamir shares).
3. **Phase 3: Equivalize, Passwords, and Concatenation.** For each value in $\mathcal{V}_A$:
   - *Equivalize*: if a value in $\mathcal{V}_P$ is structurally equivalent to a value in $\mathcal{V}_A$, add the principal's version to $\mathcal{V}_A$. This captures cases where the attacker "recognizes" a value it already knows in a different form.
   - *Passwords*: if the attacker knows a value containing a `password`-qualified constant, and the attacker can *verify a guess* for that password, then the attacker recovers the password value. This models offline dictionary attacks against weak secrets. The verification condition is defined precisely in §4.5.
   - *Concatenation*: if the attacker knows a value that reveals its arguments (e.g. the output of a primitive that passively reveals all inputs), extract those arguments.

If any phase produces new knowledge, the loop restarts from Phase 1 with the updated $\mathcal{V}_A$. When all three phases produce no new values, the loop has reached a *fixed point* and terminates. The deduction loop is a pure fixed-point computation: it does not check queries or exit early. Query evaluation occurs in a separate phase after the closure completes.

4.2.2 *Passive Attacker Analysis*

Under a passive attacker, analysis proceeds as follows for each phase declared in the model:

1. Initialize a fresh attacker state.
2. Populate $\mathcal{V}_A$ with: all public constants, all leaked constants (for the current phase), and all values exchanged on the wire.
3. For each principal, resolve all values, perform symbolic rewrites, and run the deductive analysis loop (§4.2.1).
4. Check all queries.

Since the passive attacker cannot modify messages, only a single protocol execution needs to be analyzed per phase.

4.2.3 *Active Attacker Analysis*

Under an active attacker, Verifpal must explore the space of all possible attacker mutations — that is, all the ways the attacker can substitute unguarded wire values with values it knows or can construct. This search is parameterized by a single *depth* value d (default $d = 3$), with a precise coverage guarantee:

> At depth d, Verifpal explores all attacker strategies involving at most d simultaneous value substitutions, with injected values of nesting depth at most d.

The search proceeds through increasing depth levels:

1. **Depth 0 (Baseline).** Run the deductive analysis loop on the unmodified protocol execution, exactly as in passive analysis.
2. **Depth 1 (Single-variable mutations).** Mutate each unguarded wire constant individually. All attacker-known values are available as replacements, including `nil` for constants and `G^nil` for equations (the attacker's own Diffie-Hellman public key). This naturally covers the canonical man-in-the-middle attack on DH key exchanges: replacing a received `G^b` with `G^nil` is a first-order mutation that requires no special case.
3. **Depth 2 (Pairs).** Pairs of simultaneous variable mutations, plus recursive injection nesting up to depth 1. The attacker can construct values by substituting arguments of known primitives with other known values, as long as the resulting primitive has the same *skeleton* (§4.3).
4. **Depth 3 (Triples).** Triples of simultaneous variable mutations, plus injection nesting up to depth 2.

At each depth level, for each combination of mutations, Verifpal:

1. Applies the mutations to the principal's state (replacing wire-received values).
2. Resolves all symbolic values in the mutated state.
3. Performs all symbolic rewrites. If a checked primitive fails, the execution is *truncated* at that point.
4. Runs the deductive analysis loop on the resulting state.
5. Checks all unresolved queries.

If any depth level produces new attacker knowledge, the mutation map is recalculated with the expanded set of known values and the search continues. Analysis terminates when either all queries have been resolved, or the search is *exhausted* (no new knowledge is gained at a depth level).

4.2.4 *Guard Bypass*

When a checked primitive fails during an active attacker execution — for example, because the attacker substituted a public key, causing `AEAD_DEC` to fail — Verifpal does not simply discard the execution. Instead, it checks whether the attacker can obtain the *bypass key* for the failed primitive (e.g. the symmetric key for `AEAD_DEC`, or the private signing key for `SIGNVERIF`). If the attacker possesses this key, Verifpal constructs a *bypass state* in which the attacker injects a value it controls into the failed primitive's slot, and re-resolves the principal's state. This process iterates (up to a bounded number of times) to handle cascading guard failures, where bypassing one guard reveals information needed to bypass subsequent guards.

4.3 SKELETON-BASED INJECTION

A key challenge in active attacker analysis is generating the set of values the attacker could plausibly inject in place of a legitimate value. Naively enumerating all possible combinations of known values would lead to a combinatorial explosion. Verifpal addresses this through *skeleton-based filtering*.

A *skeleton* is a normalized form of a primitive in which all "secret" components are erased: constants are replaced with `nil`, and equations are replaced with `G` or `G^nil` depending on their length. The skeleton preserves the *structural shape* of the primitive — its function name, nesting depth, and the types (constant, equation, or primitive) of each argument position.

When considering what values the attacker could inject in place of a legitimate primitive, Verifpal filters candidates through a three-step pipeline:

1. **Primitive ID check**: the candidate must use the same cryptographic function.
2. **Depth check**: the candidate's nesting depth must not exceed the original's.
3. **Skeleton hash check**: a fast structural hash is compared to reject non-matching candidates before performing the more expensive full structural comparison.

This ensures that the attacker can only inject values whose structure is compatible with what the protocol expects, dramatically reducing the search space while preserving the ability to find meaningful attacks.

4.4 PREVENTING STATE SPACE EXPLOSION

A common problem among symbolic model protocol verifiers is that for complex protocols, the space of states and value combinations that the verifier must assess becomes too large for the verifier to terminate in a reasonable time. Verifpal addresses this through several complementary techniques:

- **Bounded-depth search.** As described in §4.2.3, the depth parameter d simultaneously controls the maximum number of simultaneous mutations, the injection nesting depth,

and the k-subset weight. At each depth level, all attacker-known values are available as mutation candidates. Simple attacks are found at depth 1; more complex multi-variable attacks require higher depths.

- **Budget constraints.** Each depth level operates under explicit budget limits that cap the number of mutation combinations explored:
  - A per-principal budget of 80,000 mutation scans per depth level.
  - A cross-product cap of 50,000 combinations per variable subset.
  - A maximum of 150 subsets sampled per weight level.
- **Exhaustion detection.** If the attacker gains no new knowledge at a depth level, the search for the current phase is terminated early.
- **Skeleton filtering.** As described in §4.3, structural filtering dramatically reduces the set of candidate injection values.
- **Parallelism.** Depth levels 2 and above are executed in parallel using multi-threading, with pairs of consecutive depth levels running concurrently.
- **Bounded arity.** All primitives are limited to at most 5 inputs and 5 outputs, bounding the branching factor of the search.

Unlike ProVerif, which achieves sound verification for an unbounded number of sessions through over-approximation (at the cost of potential false positives and non-termination), Verifpal's bounded-depth search is designed to *always terminate* in reasonable time while covering the space of attacks that arise in practical protocol constructions.

4.5 PASSWORD EXTRACTION

The `password` qualifier models low-entropy secrets that are susceptible to offline dictionary attacks. Unlike `private` values (which the attacker can never guess), a `password` value can be recovered by the attacker if the attacker possesses enough context to *verify a guess*. The verification condition is:

> A password at position i of a primitive $P(a_0, \ldots, a_n)$ known to the attacker is **obtainable** if and only if:
>
> 1. Position i is not in P's `password_hashing` list, **and**
> 2. the attacker knows every sibling argument a_j for all $j \neq i$.

Condition 1 captures *inherent* computational resistance: `PW_HASH` is the only primitive whose `password_hashing` list is non-empty, reflecting the fact that password hashing functions (e.g. bcrypt, Argon2) are designed to resist brute-force even when the attacker has the hash output and full knowledge of all inputs.

Condition 2 models the *offline verification oracle*: to confirm a password guess, the attacker must be able to reconstruct the full primitive by substituting their guess at position i while supplying

the remaining arguments from their own knowledge. If any sibling argument is unknown, the attacker cannot distinguish a correct guess from an incorrect one.

Both conditions are evaluated at every primitive level in the nesting chain. For a password nested inside $P_1(P_2(\ldots, pwd, \ldots), \ldots)$, the attacker must know the siblings at *every* enclosing primitive to verify a guess. Concretely:

- `ENC(pwd, secret)`: the attacker has the ciphertext but does not know `secret`. Without the expected plaintext, the attacker cannot verify whether a candidate key decrypts to a meaningful result. The password is safe.
- `ENC(pwd, pubdata)`: the attacker knows `pubdata`. For each candidate password, the attacker computes `ENC`(*guess*, `pubdata`) and compares against the known ciphertext. The password is obtainable.
- `ENC(key, CONCAT(pwd, data))`: the password is nested inside `CONCAT` inside `ENC`. Even at the `CONCAT` level, the attacker must know `data`; at the `ENC` level, the attacker must know `key`. If either is unknown, the password is safe.
- `PW_HASH(pwd)`: regardless of the attacker's knowledge, the password is protected by Condition 1 (`password_hashing` covers all positions of `PW_HASH`).

Because the password extraction rule runs inside the fixed-point deduction loop (§4.2.1), it interacts naturally with other rules. If the attacker initially lacks a sibling argument but later obtains it through decomposition or reconstruction, the next iteration of the loop will re-evaluate the password and extract it. This captures cascading attacks where one leaked value enables brute-forcing another.

4.6 SOUNDNESS AND COMPLETENESS

Verifpal has been used to analyze TLS, Signal, Scuttlebutt, Telegram, ProtonMail and other protocols. In all cases, its results have been consistent with prior analyses of these protocols. While this empirical track record provides confidence, it is important to state precisely what Verifpal's analysis can and cannot guarantee.

4.6.1 *What Verifpal Guarantees*

Verifpal provides the following guarantee:

- **Attack soundness.** If Verifpal reports a contradiction to a query, the reported attack is a genuine logical consequence of the model. There are no "false positives" in the sense of fabricated attacks: every reported attack corresponds to a concrete sequence of attacker actions (mutations, decompositions, reconstructions) that leads to the query being violated. The attack trace produced by Verifpal constitutes a witness to the vulnerability.

More precisely:

- If a confidentiality query is contradicted, the attacker has derived a value equivalent to the protected value through legitimate application of the deduction rules to values it legitimately obtained.
- If an authentication query is contradicted, there exists a protocol execution in which the recipient uses the queried value in a primitive, but the value's sender does not match the sender specified in the query.
- If an equivalence query is contradicted, there exists a protocol execution in which the queried values resolve to structurally distinct terms.

4.6.2 *What Verifpal Does Not Guarantee*

Verifpal does *not* provide the following:

- **Verification completeness.** The absence of a reported attack does not constitute a proof that no attack exists. Verifpal's bounded search may fail to explore the specific mutation combination that would reveal a vulnerability. In formal methods terminology, Verifpal is a *bug-finding* tool, not a *proof-producing* verifier.
- **Unbounded session analysis.** Verifpal analyzes a single protocol scenario (one execution trace with attacker mutations), not an arbitrary number of concurrent sessions. Attacks that require interleaving multiple honest sessions or exploiting session-identifier confusion are outside Verifpal's scope.
- **Computational soundness.** Verifpal operates in the symbolic (Dolev-Yao) model, where cryptographic primitives are idealized. Attacks that exploit concrete cryptographic weaknesses (e.g. padding oracles, timing side channels, weak random number generation) are not modeled.

4.6.3 *The Deduction Fixed Point*

The deduction loop (§4.2.1) does provide a local completeness property: for a *given* protocol execution (i.e. a fixed set of mutations), the loop computes the *complete* set of values derivable by the attacker through the supported deduction rules (decomposition, reconstruction, recomposition, equivalization). That is, once the loop reaches its fixed point, no further applications of these rules can yield new values. The incompleteness arises from the bounded exploration of *which* mutations to try, not from the deduction within each execution.

4.6.4 *Practical Significance*

Our central argument is that the analysis logic described in this chapter, combined with the bounded-depth search, is sufficient to capture the vast majority of confidentiality and authentication attacks that arise in practical protocol constructions. The depth-based search naturally prioritizes mutation patterns that correspond to real-world attacks: single-variable key substitutions (depth 1, including DH man-in-the-middle), pairs of simultaneous mutations (depth 2), and triples with recursive injection (depth 3). The default depth of 3 finds all attacks in published protocols (TLS 1.3, Signal, Noise, Scuttlebutt, DP-3T) within seconds. Verifpal's empirical track record across well-studied protocols provides evidence for this claim.

4.6.5 *Value Construction*

Protocol analysis always begins from the point of view of the attacker. The initial set of values that the attacker can know are necessarily constants, since only constants can be exchanged within network messages (Figure 1). *"Pure"* constants (constants that are declared via a `knows` or `generates` expression and not via assignment) resolve to themselves ($x \rightarrow x$). Assigned constants resolve to either a primitive or an equation. Primitives can take constants, primitives or equations as arguments but always return constants. Equations can only take constants as arguments (effectively exponents).

4.6.6 *Deconstructions, Rewrites, and Checks*

Verifpal primitives have two kinds of potential rules:

- **Decomposition rules** allow principals and the attacker to obtain the value of a primitive's argument by knowing the primitive's output and only some of the primitive's other arguments.
  For example, knowing `e = ENC(k, m)` and `k` allows us to obtain `m`. `AEAD_ENC`, `AEAD_DEC`, `ENC`, `DEC`, `PKE_ENC`, `PKE_DEC`, and `BLIND` have decomposition rules.
- **Rewrite rules** allow principals and the attacker to rewrite a primitive's assigned value if certain conditions are satisfied.
  For example, `d = AEAD_DEC(k, e, a)` would be rewritten to `d = p` if `e = AEAD_ENC(k, p, a)`. When we *"check"* a primitive (see §2.3.2), a failed rewrite is essentially what we are terming as a *"failed check"* — checks make it such that failed rewrites truncate session execution at that point. `ASSERT`, `SPLIT`, `SIGNVERIF`, `DEC`, `AEAD_DEC`, `PKE_DEC`, `RINGSIGNVERIF`, and `UNBLIND` have rewrite rules.

4.6.7 *Provenance of Values*

In Verifpal, once a constant is known, generated or assigned, a *provenance* record is attached to it, tracking the principal responsible for creating it (*creator*), who sent it (*sender*), and whether the attacker has tampered with it (*attacker_tainted*). For example, if Alice creates a value `m` and sends it to Bob, then `m` has Alice as both its creator and sender in Bob's state. If the active attacker replaces `m` on the wire, the sender becomes the attacker and the value is marked as tainted.

When an attacker is tasked with contradicting an authentication query, it checks whether the provenance of the queried value indicates that it was sent by a principal other than the one claimed by the query, or that the attacker injected a replacement value that was subsequently used in a cryptographic operation.

4.6.8 *Value Resolution and Trust*

A subtle but critical aspect of Verifpal's analysis is how it resolves symbolic values within a principal's state. Each slot in a principal's state tracks three versions of a value:

- *original*: the value as originally computed by the protocol, before the attacker tampered with it.

- *pre_rewrite*: the value after attacker mutations but before symbolic rewrites.
- *value*: the final canonical value after both mutations and rewrites.

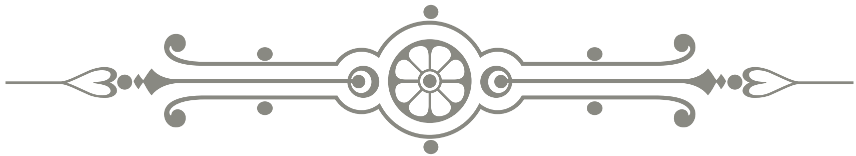
When resolving nested expressions, Verifpal must decide which version to use. The decision is based on the value's provenance: if the value is *not tainted* by the attacker, or the principal *created it themselves*, or the principal *did not receive it over a wire*, the *original* value is used. Only for values that were received over the wire and tainted by the attacker is the *value* (the attacker's version) used. This ensures that the analysis accurately reflects what each principal would actually compute given the attacker's manipulations.

4.6.9 *Mutations and Guarded Constants*

Except for guarded constants, the active attacker can substitute any unguarded wire constant with any value it knows or can construct. The goal of these substitutions is to explore protocol executions under different attacker-controlled inputs. Each unguarded constant may be replaced with:

- **Other constants and values from the protocol** that have been revealed to the attacker.
- **New primitive expressions** constructed from attacker-known values, subject to skeleton compatibility constraints (§4.3).
- **Malicious values** crafted by the attacker, including for example malicious public keys (`G^nil`) or malicious signatures under key pairs generated and owned by the attacker.

Whenever the attacker gains new values through this process, the mutation space is recalculated with the expanded knowledge and subsequent depth levels explore new combinations. Analysis for a given phase ends when all queries are resolved, or when the search is exhausted (no new knowledge gained at a depth level).



In Part II of this manual, we will look at how popular secure protocols such as Signal and Scuttlebutt, can be modeled in Verifpal. We will go through the rationale behind the construction of the model and queries and the capabilities given to the attacker. Finally, we will cover the results of Verifpal's analysis and see if it changes based on how we tweak the model. By looking at these three protocols, you will hopefully obtain a more complete picture on verification with Verifpal.

PART II

Protocol Examples in Verifpal

CHAPTER 5 SECURE MESSAGING WITH SIGNAL

Introduced in 2014, the Signal protocol[1] started off as the core of the eponymous Signal messaging app for Android and iOS devices. In the following years it was also adopted by WhatsApp, Facebook Messenger, Skype and other applications. Today, it is responsible for encrypted communications on at least a billion devices worldwide, competing with Apple's iMessage protocol and Telegram's MTProto protocol[2].

5.1 SECURITY GOALS

Aside from targeting obvious security goals such as message confidentiality and mutual authentication for principals, Signal differentiated itself from predecessors as well as from its competitor protocols by offering some ambitious security properties. The core design element behind these features is the fact that in Signal, each principal has essentially two types of key pairs: *long-term key pairs*, which serve to authenticate the identity of Alice and Bob to one another, are used exclusively for signing and for session establishment and that never change, and *ephemeral key pairs*, which last at most for a handful of messages and are used solely for encryption. The point of this approach is target the following security goals:

- *Forward-secure authenticated key exchange.* After a Signal session is established between Alice and Bob, revealing any or both parties' long-term keys does not reveal the contents of any of their messages[3]. Since long-term keys are the only key material that remains on-device for extended periods of time, it can be assumed that this security goal is supposed to guard against device theft.
- *Per-message forward secrecy and post-compromise security.* If Alice or Bob's state were to be compromised at any point in time, the number of past and future messages, relevant

[1] `https://signal.org/docs/`

[2] We focus on Signal as an example in this manual because it achieves stronger security properties than iMessage and MTProto.

[3] This property is not specifically new to Signal, but was also used by the Off-the-Record messaging protocol, first presented in 2004.

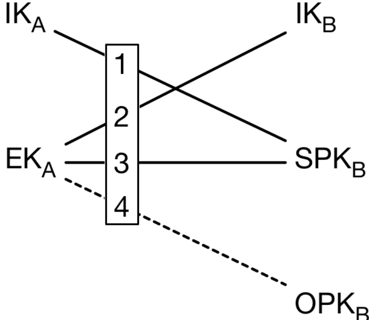


Figure 5.1: Signal's *"X3DH"* authenticated key exchange. IK_A and IK_B represent Alice and Bob's long-term key pairs. EK_A represents Alice's ephemeral session key pair. SPK_B and OPK_B represent Bob's signed ephemeral pre-key and one-time ephemeral pre-key. Three Diffie-Hellman shared secret calculations, and one optional Diffie-Hellman shared secret calculation, are conducted.

to the last message sent at time of compromise, is limited[4].

Aside from these security-centric features, Signal also offers *asynchronous (*"offline"*) session establishment:* Alice is able to establish a Signal session with Bob and send a message even if Bob's phone is turned off. When Bob turns his phone back on, he will immediately receive Alice's message (even if, at the time, Alice's phone is off.) This mirrors the behavior of SMS, which people are likely to expect on mobile devices. This SMS-like use case significantly affects Signal's design.

5.2 PRINCIPALS

Our first step in Verifpal will be to model Signal's essential protocol components and then to illustrate how these components can be used by Alice and Bob in order to conduct a Signal session.

5.2.1 *Modeling the Key Exchange*

Figure 5.1 illustrates how Signal's authenticated key exchange works. When initiating a session with Bob, Alice will perform four Diffie-Hellman operations:

1. Between Alice's long-term private key and Bob's *"signed pre-key"*, an ephemeral public key that Bob has pre-emptively generated, signed using his long-term private key, and stored on the Signal server.
2. Between Alice's ephemeral private key, generated for this session, and Bob's long-term public key.
3. Between Alice's ephemeral private key and Bob's signed pre-key.

[4] *How* limited is a matter of debate. While the Signal protocol tries to enforce this property between every message, real-world considerations such as network unreliability makes this practically impossible to maintain, and applications such as WhatsApp can have significantly wide forward secrecy *"windows of compromise"* enveloping multiple messages.

4. Between Alice's ephemeral private key and Bob's *"one-time pre-key"*, an ephemeral public key that Bob has pre-emptively generated and stored on the Signal server. Unlike the signed pre-key, it is not signed[5].

The four values obtained above are then hashed into a single value known as the master secret. Alice can also include an encrypted message along with her key exchange message, therefore accomplishing the "SMS-like" behavior mentioned earlier. So, let's declare Alice and Bob in Verifpal:

Signal: Initializing Alice

```
^^Iattacker[active]
^^Iprincipal Alice[
^^I^^I^^Iknows public c0, c1, c2, c3, c4
^^I^^I^^Iknows private alongterm
^^I^^I^^Igalongterm = G^alongterm
^^I^^I]
```

Signal: Initializing Bob

```
^^Iprincipal Bob[
^^I^^I^^Iknows public c0, c1, c2, c3, c4
^^I^^I^^Iknows private blongterm, bs
^^I^^I^^Igenerates bo
^^I^^I^^Igblongterm = G^blongterm
^^I^^I^^Igbs = G^bs
^^I^^I^^Igbo = G^bo
^^I^^I^^Igbssig = SIGN(blongterm, gbs)
^^I^^I]
```

Now, let's have Alice initiate a session with Bob and derive a master secret, which she stores as `amaster`:

Signal: Alice Initiates Session with Bob

```
^^IBob → Alice: [gblongterm], gbssig, gbs, gbo
^^Iprincipal Alice[
^^I^^I^^Igenerates ae1
^^I^^I^^Igae1 = G^ae1
^^I^^I^^Iamaster = HASH(c0, gbs^alongterm, gblongterm^ae1, gbs^ae1, gbo^ae1)
^^I^^I^^Iarkba1, ackba1 = HKDF(amaster, c1, c2)
^^I^^I]
```

[5] Signed pre-keys are rotated roughly once a week, while one-time pre-keys are only used once. This is simply because one-time pre-keys are consumed after a single use, and having Bob's phone sign each one (of which a server could store hundreds at a time) would add overhead with limited benefit, since the authentication of one-time pre-keys ultimately derives from the key exchange structure rather than from individual signatures.

5.2.2 *Modeling Messages and the Double Ratchet*

Since long-term keys are only employed in master secret derivation, and since we want to achieve per-message forward secrecy and post-compromise security, we want to both *authenticate* future messages based on Alice and Bob's identities while keeping them *confidential* using perpetually fresh ephemeral shared secrets. The logic behind this *"Double Ratchet"* mechanism is fairly complicated, but in essence, here's how we can model it in Verifpal:

Signal: Alice Encrypts Message 1 to Bob

```
^^Iprincipal Alice[
^^I^^I^^Igenerates m1, ae2
^^I^^I^^Igae2 = G^ae2
^^I^^I^^Ivalid = SIGNVERIF(gblongterm, gbs, gbssig)?
^^I^^I^^Iakshared1 = gbs^ae2
^^I^^I^^Iarkab1, ackab1 = HKDF(akshared1, arkba1, c2)
^^I^^I^^Iakenc1, akenc2 = HKDF(MAC(ackab1, c3), c1, c4)
^^I^^I^^Ie1 = AEAD_ENC(akenc1, m1, HASH(galongterm, gblongterm, gae2))
^^I^^I]
^^IAlice → Bob: [galongterm], gae1, gae2, e1
```

Notice how Alice generates a second fresh ephemeral key pair, `(ae2, gae2 = G^ae2)`, and mixes it with the master secret in order to derive two symmetric keys, `ackab1` will be used for encryption, while `arkab1` will only be used to derive future pairs of symmetric keys in the same fashion, thereby keeping a relationship back to the master secret, which ensures that all future derived keys are mixed with the key material that provided authentication in the master secret.

Notice also how the `SIGNVERIF` primitive is checked — if Alice can't verify the signature of Bob's signed pre-key `gbs` using Bob's long-term signing public key `gblongterm`, then the entire session is aborted.

Finally, notice how we are guarding `gblongterm` and `galongterm` from being modified by an active attacker while in transit – this achieves a model where Alice and Bob have mutually pre-authenticated one another's long-term public keys.

Alice then encrypts her chosen plaintext message `m1` to produce ciphertext `e1`. Notice how the Signal protocol specifies that a hash of the public keys used in this session must go as associated data to the message encryption primitive. This helps achieve a property known as *session* or *channel binding*.

Here's how Bob can decrypt Alice's first message (after also generating the master secret):

Signal: Bob Derives Shared Master Secret

```
^^Iprincipal Bob[
^^I^^I^^Ibmaster = HASH(c0, galongterm^bs, gae1^blongterm, gae1^bs, gae1^bo)
^^I^^I^^Ibrkba1, bckba1 = HKDF(bmaster, c1, c2)
^^I^^I]
```

Signal: Bob Decrypts Alice's Message 1

```
^^Iprincipal Bob[
^^I^^I^^Ibkshared1 = gae2^bs
^^I^^I^^Ibrkab1, bckab1 = HKDF(bkshared1, brkba1, c2)
^^I^^I^^Ibkenc1, bkenc2 = HKDF(MAC(bckab1, c3), c1, c4)
^^I^^I^^Im1_d = AEAD_DEC(bkenc1, e1, HASH(galongterm, gblongterm, gae2))
^^I^^I]
```

And here's how Bob can send his reply, encrypting his message `m2` to produce ciphertext `e2`. Notice how with each message, a new key pair is generated and mixed in with the chain of keys continuously descending from the master secret — that's what Signal's Double Ratchet is all about:

Signal: Bob Encrypts Message 2 to Alice

```
^^Iprincipal Bob[
^^I^^I^^Igenerates m2, be
^^I^^I^^Igbe = G^be
^^I^^I^^Ibkshared2 = gae2^be
^^I^^I^^Ibrkba2, bckba2 = HKDF(bkshared2, brkab1, c2)
^^I^^I^^Ibkenc3, bkenc4 = HKDF(MAC(bckba2, c3), c1, c4)
^^I^^I^^Ie2 = AEAD_ENC(bkenc3, m2, HASH(gblongterm, galongterm, gbe))
^^I^^I]
^^IBob → Alice: gbe, e2
```

For good measure, we model a final message `m3` sent from Alice to Bob, after Alice decrypts Bob's message:

Signal: Alice Decrypts Message 2

```
^^Iprincipal Alice[
^^I^^I^^Iakshared2 = gbe^ae2
^^I^^I^^Iarkba2, ackba2 = HKDF(akshared2, arkab1, c2)
^^I^^I^^Iakenc3, akenc4 = HKDF(MAC(ackba2, c3), c1, c4)
^^I^^I^^Im2_d = AEAD_DEC(akenc3, e2, HASH(gblongterm, galongterm, gbe))
^^I^^I]
```

Signal: Alice Encrypts Message 3 to Bob

```
^^Iprincipal Alice[
^^I^^I^^Igenerates m3, ae3
^^I^^I^^Igae3 = G^ae3
^^I^^I^^Iakshared3 = gbe^ae3
^^I^^I^^Iarkab3, ackab3 = HKDF(akshared3, arkba2, c2)
^^I^^I^^Iakenc5, akenc6 = HKDF(MAC(ackab3, c3), c1, c4)
^^I^^I^^Ie3 = AEAD_ENC(akenc5, m3, HASH(gblongterm, galongterm, gae3))
^^I^^I]
^^IAlice → Bob: gae3, e3
```

Signal: Bob Decrypts Message 3

```
^^Iprincipal Bob[
^^I^^I^^Ibkshared3 = gae3^be
^^I^^I^^Ibrkab3, bckab3 = HKDF(bkshared3, brkba2, c2)
^^I^^I^^Ibkenc5, bkenc6 = HKDF(MAC(bckab3, c3), c1, c4)
^^I^^I^^Im3_d = AEAD_DEC(bkenc5, e3, HASH(gblongterm, galongterm, gae3))
^^I^^I]
```

Finally, we want to specify that, at a later point in time after their session has been conducted, it is possible that Alice and Bob will both have their phones stolen, thereby revealing their long-term private keys (but not their ephemeral private keys) to the attacker. We uses phases (as described in §2.5) to express this:

Signal: Long-Term Private Key Leakage in Subsequent Phase

```
^^Iphase[1]

^^Iprincipal Alice[leaks alongterm]
^^Iprincipal Bob[leaks blongterm]
```

Now that we've modeled a fairly illustrative and representative execution of the Signal protocol between Alice and Bob, covering an authenticated key exchange as well as three messages, we're finally ready to ask Verifpal some tough questions and to analyze if, and how, our model of Signal achieves its desired security goals.

5.3 QUERIES AND ANALYSIS

Given that Signal is a secure messaging protocol, we certainly want to check whether `m1`, `m2` and `m3` are confidential against an active attacker. We also want to check if an attacker can impersonate any of the principals in sending one of the above messages.

Formulating these queries in Verifpal is straightforward:

Signal: Message Queries

```
^^Iqueries[
^^I^^I^^Iconfidentiality? m1
^^I^^I^^Iauthentication? Alice -> Bob: e1
^^I^^I^^Iconfidentiality? m2
^^I^^I^^Iauthentication? Bob -> Alice: e2
^^I^^I^^Iconfidentiality? m3
^^I^^I^^Iauthentication? Alice -> Bob: e3
^^I^^I]
```

Now, let's look at our initial results:

Signal: Initial Analysis Results

```
^^IVerifpal! verification completed at 12:36:53
```

This indicates that Verifpal was unable to find a contradiction to any of the queries. This goes hand in hand with previous academic formal verification work on Signal [19, 20]: if Alice and Bob initiate a session with mutual pre-authentication, and if Alice is aborting the session should Bob's signed pre-key not pass signature verification, then the Signal protocol achieves confidentiality and authentication for messages sent between the two parties. Great!

If we uncheck Alice's usage of `SIGNVERIF`, we see that results don't change. But what happens if we then also unguard Bob's long-term public key as it is being sent to Alice?

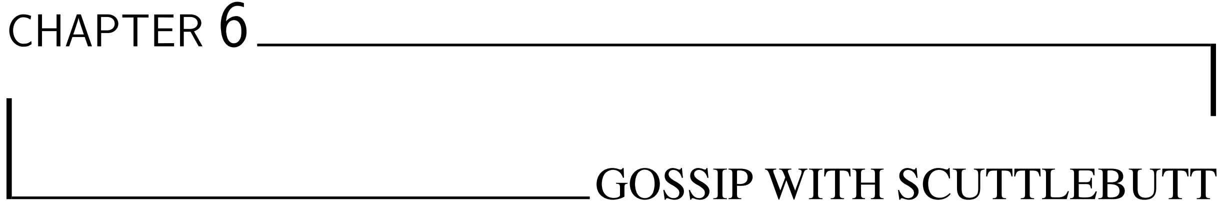
Signal: Results with Mayor-in-the-Middle on Bob's Keys

```
^^IResult • authentication? Bob→ Alice: e2: When the following values are controlled by
     Attacker:
^^Igblongterm → G^nil (originally G^blongterm)
^^Igbs → G^nil (originally G^bs)
^^Igbo → G^nil (originally G^bo)
^^Igbe → G^nil (originally G^be)
^^Ie2 (AEAD_ENC(HKDF(MAC(HKDF(G^ae2^be, HKDF(G^ae2^bs, HKDF(HASH(c0, G^alongterm^bs, G^
    ae1^blongterm, G^ae1^bs, G^ae1^bo), c1, c2), c2), c2), c3), c1, c4), m2, HASH(G^
    blongterm, G^alongterm, G^be))), sent by Bob, is successfully used in AEAD_DEC(HKDF(
    MAC(HKDF(G^nil^ae2, HKDF(G^nil^ae2, HKDF(HASH(c0, G^nil^alongterm, G^nil^ae1, G^nil^
    ae1, G^nil^ae1), c1, c2), c2), c2), c3), c1, c4), AEAD_ENC(HKDF(MAC(HKDF(G^ae2^be,
    HKDF(G^ae2^bs, HKDF(HASH(c0, G^alongterm^bs, G^ae1^blongterm, G^ae1^bs, G^ae1^bo),
    c1, c2), c2), c2), c3), c1, c4), m2, HASH(G^blongterm, G^alongterm, G^be)), HASH(G^
    nil, G^alongterm, G^nil)) within Alice's state, despite being vulnerable to
    tampering.
^^I(Analysis 25)
^^IResult • confidentiality? m1: When the following values are controlled by Attacker:
^^Igblongterm → G^nil (originally G^blongterm)
^^Igbs → G^nil (originally G^bs)
^^Igbo → G^nil (originally G^bo)
^^Igbe → G^nil (originally G^be)
^^Im1 (m1) is obtained by Attacker.
^^I(Analysis 26)
^^IResult • confidentiality? m3: When the following values are controlled by Attacker:
^^Igblongterm → G^nil (originally G^blongterm)
^^Igbs → G^nil (originally G^bs)
^^Igbo → G^nil (originally G^bo)
^^Igbe → G^nil (originally G^be)
^^Im3 (m3) is obtained by Attacker.
```

However, if we were to keep Bob's long-term public key guarded while compromising Bob's long-term *private* keys *after* the session by using phases (§2.5), we would see that forward secrecy would hold in Signal.

Tweaking your model and re-running analysis is central to getting the most insight out of Verifpal. By making some very simple changes to our model, we were quickly able to go from a fully secure model to one that showed us whether forward secrecy would be achieved in the event

of a long-term private key compromise, and then to another that provided a warning on the importance of mutual pre-authentication.



GOSSIP WITH SCUTTLEBUTT

Scuttlebutt[1] is a protocol for decentralized communication. While the full protocol includes mechanisms for many secure features, including private group chat, in this chapter we will be looking at the Scuttlebutt authenticated key exchange and seeing how we can model and analyze it in Verifpal.

6.1 SECURITY GOALS

Scuttlebutt documents a variety of security goals that the protocol aims to accomplish. In our analysis, we will focus on a handful of these goals:

- *Initiator identity hiding.* An attacker cannot learn the public key of the initiator.
- *Message confidentiality.* An attacker cannot learn the content of messages exchanged between principals.
- *Network identifier hiding.* Both peers need to know a key that represents the particular Scuttlebutt network they wish to connect to, however a mayor-in-the-middle can't learn this key from the handshake.
- *Forward secrecy.* Recording a user's network traffic and then later stealing their secret key will not allow an attacker to decrypt their past handshakes.

6.2 PRINCIPALS

Similarly to Signal, Scuttlebutt also gives each principal a long-term key pair, used for identity authentication, and ephemeral key pairs used for encryption. Let's initialize Alice and Bob's states:

[1] `https://ssbc.github.io/scuttlebutt-protocol-guide/`

Declaring New Principals: Alice and Bob

```
^^Iprincipal Alice[
^^I^^I^^Iknows public null
^^I^^I^^Iknows private n
^^I^^I^^Iknows private longTermA
^^I^^I^^Igenerates ephemeralA
^^I^^I^^IlongTermAPub = G^longTermA
^^I^^I^^IephemeralAPub = G^ephemeralA
^^I^^I]
^^Iprincipal Bob[
^^I^^I^^Iknows public null
^^I^^I^^Iknows private n
^^I^^I^^Iknows private longTermB
^^I^^I^^Igenerates ephemeralB
^^I^^I^^IlongTermBPub = G^longTermB
^^I^^I^^IephemeralBPub = G^ephemeralB
^^I^^I]
^^IBob → Alice: [longTermBPub]
```

Note that in the above, we are declaring `n`, the so-called Scuttlebutt *"network identifier"*, to be a private pre-known value, unknown to the attacker. It is not clear how realistic this model is, as the Scuttlebutt protocol seems to expect all users of a network to know this value, but for it to be simultaneously unknown to an attacker. We'll see later what changes if we re-run our analysis with `n` being a publicly known value.

Unlike Signal, Scuttlebutt's key exchange is rather wordy and takes its time, spanning over two round trips. In the first round trip, Alice and Bob simply exchange client and server *"hello"* messages:

Scuttlebutt: Alice and Bob Exchange Ephemeral Public Keys

```
^^Iprincipal Alice[
^^I^^I^^InMacAlice = MAC(n, ephemeralAPub)
^^I^^I]
^^IAlice → Bob: ephemeralAPub, nMacAlice
^^Iprincipal Bob[
^^I^^I^^InMacAliceValid = ASSERT(MAC(n, ephemeralAPub), nMacAlice)?
^^I^^I^^InMacBob = MAC(n, ephemeralBPub)
^^I^^I]
^^IBob → Alice: ephemeralBPub, nMacBob
```

The goal of the `MAC` here is simply to provide *context* or *channel binding* to the generated values, so as to avoid them being re-usable by an attacker in a different Scuttlebutt network, which would have a different identifier[2].

Alice then proceeds to generate two session secrets: one that she uses to encrypt her long-term public key to Bob (thereby hiding it from the attacker), and another that she will use to encrypt messages:

[2]Again, it is unclear how seriously we can expect a strong attacker not to know the identifier of the networks they are attempting to conduct active attacks in, but that is not something we can decide.

Scuttlebutt: Alice Generates Session Secrets

```
^^Iprincipal Alice[
^^I^^I^^InMacBobValid = ASSERT(MAC(n, ephemeralBPub), nMacBob)?
^^I^^I^^IephemeralSecretAlice = ephemeralBPub^ephemeralA
^^I^^I^^IlongTermSecretAlice = longTermBPub^ephemeralA
^^I^^I^^ImasterSecret1Alice = HASH(n, ephemeralSecretAlice, longTermSecretAlice)
^^I^^I^^Isig1Alice = SIGN(longTermA, HASH(n, longTermBPub, ephemeralSecretAlice))
^^I^^I^^IsecretBox1Alice = AEAD_ENC(masterSecret1Alice, sig1Alice, null)
^^I^^I^^IsecretBox2Alice = AEAD_ENC(masterSecret1Alice, longTermAPub, null)
^^I^^I^^IlongEphemeralSecretAlice = ephemeralBPub^longTermA
^^I^^I^^ImasterSecret2Alice = HASH(n, ephemeralSecretAlice, longTermSecretAlice,
    longEphemeralSecretAlice)
^^I^^I]
^^IAlice → Bob: secretBox1Alice, secretBox2Alice
```

Bob decrypts Alice's long-term public key and generates the same set of shared secrets:

Scuttlebutt: Bob Generates Session Secrets

```
^^Iprincipal Bob[
^^I^^I^^IephemeralSecretBob = ephemeralAPub^ephemeralB
^^I^^I^^IlongTermSecretBob = ephemeralAPub^longTermB
^^I^^I^^ImasterSecret1Bob = HASH(n, ephemeralSecretBob, longTermSecretBob)
^^I^^I^^Isig1Bob = AEAD_DEC(masterSecret1Bob, secretBox1Alice, null)?
^^I^^I^^IlongTermAPub_Bob = AEAD_DEC(masterSecret1Bob, secretBox2Alice, null)?
^^I^^I^^Isig1Valid = SIGNVERIF(longTermAPub_Bob, HASH(n, longTermBPub,
    ephemeralSecretBob), sig1Bob)?
^^I^^I^^IlongEphemeralSecretBob = longTermAPub_Bob^ephemeralB
^^I^^I]
```

Bob then generates and encrypts a signature confirming his intent to engage with Alice in this session:

Scuttlebutt: Bob Signs Session Transcript

```
^^Iprincipal Bob[
^^I^^I^^Isig2Bob = SIGN(longTermB, HASH(n, sig1Bob, longTermAPub_Bob, ephemeralSecretBob
    ))
^^I^^I^^ImasterSecret2Bob = HASH(n, ephemeralSecretBob, longTermSecretBob,
    longEphemeralSecretBob)
^^I^^I^^IsecretBox1Bob = AEAD_ENC(masterSecret2Bob, sig2Bob, null)
^^I^^I]
^^IBob → Alice: secretBox1Bob
```

Finally, Alice and Bob can now exchange some test messages. We use `m1` and `m2`, similar to our model of Signal:

Scuttlebutt: Alice Encrypts and Sends Message to Bob

```
^^Iprincipal Alice[
^^I^^I^^Iknows private m1
^^I^^I^^Isig2Alice = AEAD_DEC(masterSecret2Alice, secretBox1Bob, null)?
^^I^^I^^Isig2Valid = SIGNVERIF(longTermBPub, HASH(n, sig1Alice, longTermAPub,
    ephemeralSecretAlice), sig2Alice)?
^^I^^I^^IsecretBoxM1Alice = AEAD_ENC(masterSecret2Alice, m1, null)
^^I^^I]
^^IAlice → Bob: secretBoxM1Alice
```

Scuttlebutt: Bob Receives and Decrypts Message from Alice

```
^^Iprincipal Bob[
^^I^^I^^Iknows private m2
^^I^^I^^Im1Bob = AEAD_DEC(masterSecret2Bob, secretBoxM1Alice, null)?
^^I^^I^^IsecretBoxM2Bob = AEAD_ENC(masterSecret2Bob, m2, null)
^^I^^I]
```

Scuttlebutt: Bob Encrypts and Sends Message to Alice

```
^^IBob → Alice: secretBoxM2Bob
^^Iprincipal Alice [
^^I^^I^^Im2Alice = AEAD_DEC(masterSecret2Alice, secretBoxM2Bob, null)?
^^I^^I]
```

Now that we've modeled a fairly illustrative and representative execution of the Scuttlebutt protocol between Alice and Bob, covering an authenticated key exchange as well as three messages, we're finally ready to ask Verifpal some tough questions and to analyze if, and how, our model of Scuttlebutt achieves its desired security goals.

6.3 QUERIES AND ANALYSIS

Earlier in this chapter, we identified four security goals that we wanted to test for. Let's summarize them again with regards to our model. The attacker should not be able to:

- Know the initiator (Alice's) public key `longTermAPub`.
- Know messages `m1` and `m2`.
- Know the *"network identifier"* `n`.
- Know the content of messages even if long-term private keys are leaked.

Here are these security goals as Verifpal queries (with the addition of some standard authentication queries for messages:)

Scuttlebutt: Confidentiality and Authentication Queries

```
^^Iqueries[
^^I^^I^^Iconfidentiality? m1
^^I^^I^^Iconfidentiality? m2
^^I^^I^^Iconfidentiality? longTermAPub
^^I^^I^^Iauthentication? Alice→ Bob: secretBox1Alice
^^I^^I^^Iauthentication? Alice→ Bob: secretBox2Alice
^^I^^I^^Iauthentication? Bob→ Alice: secretBox1Bob
^^I^^I^^Iauthentication? Alice→ Bob: secretBoxM1Alice
^^I^^I^^Iauthentication? Bob→ Alice: secretBoxM2Bob
^^I^^I]
```

Now, let's look at our initial results:

Scuttlebutt: Initial Results

```
^^IVerifpal! verification completed at 15:24:51
```

No contradictions to our queries are found — but similarly to our initial analysis of Signal in Chapter 5, this is due to the fact that we made sure to guard Bob's long-term key and to check all signature verification primitives. So, let's unguard `longTermBPub` as it is being sent to Alice, and try again:

Scuttlebutt: Results with Mayor-in-the-Middle Attack on Bob

```
^^IVerifpal! verification completed at 15:27:27
```

No change! This might be surprising at first: can't the attacker impersonate Bob at this point? Indeed they can — but don't forget that the *"network identifier"* `n`, which is used to derive encryption keys, is considered unknown to the attacker here. It therefore acts as a *pre-shared key*[3]. Making `n` public to the attacker, therefore, coupled with unguarding Bob's long-term public key, makes a huge difference:

[3]Pre-shared keys are a common component in protocols. They usually are simply an encryption key that is considered to be privately known to the principals before the session begins. This differs from mutual pre-authentication in that pre-shared keys are symmetric keys and not public keys.

Scuttlebutt: Results with Public n and Bob MitM

```
^^IResult! confidentiality? n: n is obtained by the attacker as n
^^IResult! confidentiality? longtermapub: longtermapub is obtained by the attacker as
    longtermapub
^^IResult! authentication? Alice → Bob: secretbox1alice: secretbox1alice, sent by
    Attacker and not by Alice and resolving to AEAD_ENC(mastersecret1alice, sig1alice,
    null), is used in primitive AEAD_DEC(mastersecret1bob, secretbox1alice, null) in Bob
    's state
^^IResult! authentication? Alice → Bob: secretbox2alice: secretbox2alice, sent by
    Attacker and not by Alice and resolving to AEAD_ENC(mastersecret1alice, longtermapub
    , null), is used in primitive AEAD_DEC(mastersecret1bob, secretbox2alice, null) in
    Bob's state
```

Aside the obvious first result, we see that the attacker was able to decrypt initiator Alice's long-term public key as well as impersonate Alice to Bob in sending the first two messages. Let's guard Bob's long-term public key again, leave `n` as public, and leak Alice's long-term private key post-handshake, right after she sends `m1`:

Scuttlebutt: Alice Leaks Long-Term Private Key

```
^^IAlice → Bob: secretBoxM1Alice, longTermA
```

Aside from obtaining `n` and `longTermAPub` (since the first is public and since we leaked the private key of the second), the attacker is not able to contradict any other queries, thereby indicating forward secrecy:

Scuttlebutt: Results Showing Forward Secrecy

```
^^IResult! confidentiality? n: n is obtained by the attacker as n
^^IResult! confidentiality? longtermapub: longtermapub is obtained by the attacker as
    longtermapub
```

Tweaking your model and re-running analysis is central to getting the most insight out of Verifpal. By making some very simple changes to our model, we were quickly able to go from a fully secure model to one that showed us the security of the protocol when confronted with no authentication for the responder (Bob) with and without a pre-shared key (`n`), and then whether forward secrecy would be achieved in the event of a long-term private key compromise.

CHAPTER 7 CONTACT TRACING WITH DP-3T

This chapter is contributed by Georgio Nicolas.

In early 2020, numerous researchers published a new protocol which aimed to provide a proximity-tracking solution that can help during pandemics while still being privacy-preserving: the result, Decentralized Privacy-Preserving Proximity Tracing [21] (DP-3T), provided a promising first step in bringing real-world cryptography into the effort to combat the COVID-19 pandemic, even being eventually adopted by Apple and Google into their smartphone operating systems.

7.1 SECURITY GOALS

DP-3T's security goals are summarized thus in the DP-3T whitepaper:

> *"…to simplify and accelerate the process of identifying people who have been in contact with an infected person, thus providing a technological foundation to help slow the spread of the SARS-CoV-2 virus. The system aims to minimise privacy and security risks for individuals and communities and guarantee the highest level of data protection."*

7.2 MODELING DP-3T

To demonstrate DP-3T, we will assume that the principals participating in this simulation are the following:

- A population of 3 individuals: Alice, Bob, and Charlie, each of them possessing a smartphone: SmartphoneA, SmartphoneB, and SmartphoneC respectively;
- A Healthcare Authority serving this population;
- A Backend Server, that individuals can communicate with to obtain daily information.

After installing Verifpal, we can start by creating a new model called “dp-3t.vp” in which we begin by defining an attacker which matches with our security model. In this case we will be using an active attacker (i.e. one that can not only monitor but also intercept and overwrite unprotected messages on the network):

DP-3T: Declaring the Attacker

```
^^Iattacker[active]
```

We then proceed to illustrate our model as a sequence of days in which DP-3T is in operation within the lifecycle of a pandemic.

7.2.1 *Day 0: Setup Phase*

We assume that no new individuals were diagnosed with the disease on Day 0 of using DP-3T. This means that the Healthcare Authority and the Backend Server will not act at this stage and we can simply ignore them for now.

The DP-3T specification states that every principal, when first joining the system, should generate a random secret key (`SK`) to be used for one day only. For every `SK` value, and the knowledge of a public “broadcast key” value, principals should compute multiple Unique Ephemeral ID values (`EphID`) using a combination of a PRG and a PRF. The method of generating `EphID` is analogous with the `HKDF` function from Verifpal. We could add the following lines of code to our file in order to model Alice’s `SmartphoneA`:

DP-3T: SmartphoneA Setup

```
^^I// A principal block looks like the following
^^Iprincipal SmartphoneA[
^^I^^I^^I// In the line below we state that Alice knows the public BroadcastKey

^^I^^I^^Iknows public BroadcastKey

^^I^^I^^I// SK is going to be a secret random value
^^I^^I^^I// To define it we use the "generates" keyword
^^I^^I^^I// We will use the following template for SK variable names
^^I^^I^^I// SK[day number][principal initial]

^^I^^I^^Igenerates SK0A

^^I^^I^^I// We will use the following template for EphID variable names
^^I^^I^^I// EphID[day number][value number][principal initial]

^^I^^I^^IEphID00A, EphID01A, EphID02A = HKDF(nil, SK0A, BroadcastKey)
^^I^^I]
```

The same thing goes for Bob, and Charlie:

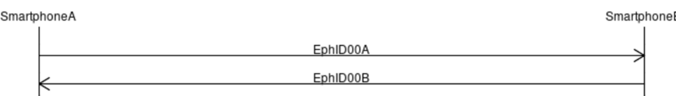
DP-3T: SmartphoneB, SmartphoneC Setup

```
^^Iprincipal SmartphoneB[
^^I^^I^^Iknows public BroadcastKey
^^I^^I^^Igenerates SK0B
^^I^^I^^IEphID00B, EphID01B, EphID02B = HKDF(nil, SK0B, BroadcastKey)
^^I^^I]

^^Iprincipal SmartphoneC[
^^I^^I^^Iknows public BroadcastKey
^^I^^I^^Igenerates SK0C
^^I^^I^^IEphID00C, EphID01C, EphID02C = HKDF(nil, SK0C, BroadcastKey)
^^I^^I]
```

Whenever two principals would come be in physical proximity of each other, they would automatically exchange `EphIDs`. Once a principal uses an `EphID` value, they discard it and use another one when performing an exchange with another principal.

Let's imagine that Alice and Bob came into contact. It would mean that Alice sent `EphID00A` in a message to Bob and that Bob sent `EphID00B` to Alice:



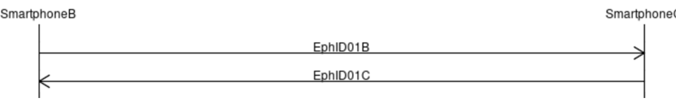
Here is how the above message exchange is modeled in Verifpal:

DP-3T: EphID Communication

```
^^I// Sender → Recipient : Name of Value

^^ISmartphoneA → SmartphoneB: EphID00A
^^ISmartphoneB → SmartphoneA: EphID00B
```

Now, let's say that in the conclusion of Day 0, Bob sits behind Charlie in the Bus:



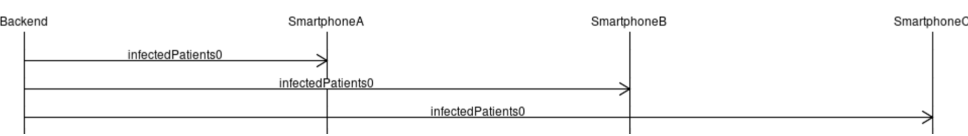
Modeling this is equally simple:

DP-3T: EphID Communication

```
^^ISmartphoneC → SmartphoneB: EphID01C
^^ISmartphoneB → SmartphoneC: EphID01B
```

7.2.2 *Day 1*

On Day 1, the Backend Server will automatically publish the `SK` values of people who were infected to the members of the general population. These values were previously unpublished and thus were private and only known by their generators and the server.



DP-3T: BackendServer Communication

```
^^Iprincipal BackendServer[
^^I^^I^^I// Let's assume that infectedPatients0 is the list of infected patients on day
    0
^^I^^I^^Iknows private infectedPatients0
^^I^^I]

^^IBackendServer → SmartphoneA: infectedPatients0
^^IBackendServer → SmartphoneB: infectedPatients0
^^IBackendServer → SmartphoneC: infectedPatients0
```

We should not forget that every day starting from Day 1, DP-3T mandates that principals will generate new `SK` values. The new value will be equal to the hash of the `SK` value from the day before. Principals will also generate `EphIDs` just like before.

DP-3T: EphID Generation

```
^^Iprincipal SmartphoneA[
^^I^^I^^ISK1A = HASH(SK0A)
^^I^^I^^IEphID10A, EphID11A, EphID12A = HKDF(nil, SK1A, BroadcastKey)
^^I^^I]

^^Iprincipal SmartphoneB[
^^I^^I^^ISK1B = HASH(SK0B)
^^I^^I^^IEphID10B, EphID11B, EphID12B = HKDF(nil, SK1B, BroadcastKey)
^^I^^I]

^^Iprincipal SmartphoneC[
^^I^^I^^ISK1C = HASH(SK0C)
^^I^^I^^IEphID10C, EphID11C, EphID12C = HKDF(nil, SK1C, BroadcastKey)
^^I^^I]
```

Thankfully, Alice, Bob and Charlie are committed to self-confinement and have stayed at home, so they did not exchange `EphIDs` with anyone.

7.2.3 *Day 2*

On Day 2, a similar sequence of events takes place. Since it is sufficient to define the values that we will need later on in our model, we will just define a block for Alice.

DP-3T: EphID Generation

```
^^Iprincipal SmartphoneA[
^^I^^I^^ISK2A = HASH(SK1A)
^^I^^I^^IEphID20A, EphID21A, EphID22A = HKDF(nil, SK2A, BroadcastKey)
^^I^^I]
```

7.2.4 *Fast-Forward to Day 15*

Unfortunately, Alice tests positive for COVID-19. Since this breaks the routine that happened between Day 1 and Day 15, we will announce a new phase[1] in our protocol model:

DP-3T: Declaring a New Phase

```
^^Iphase[1]
```

Alice decides to announce her infection anonymously using DP-3T. This means that she will have to securely communicate `SK1A` (her `SK` value from 14 days ago) to the Backend Server, using a unique trigger token provided by the healthcare authority. Assuming that the Backend Server and the Healthcare Authority share a secure connection, and that a key `ephemeral_sk` has been exchanged off the wire by the Healthcare Authority, Alice, and the Backend Server, the Healthcare Authority will encrypt a freshly generated triggerToken using `ephemeral_sk` and send it to both Alice and the Backend Server.

DP-3T: Sending Tokens to HealthCareAuthority

```
^^Iprincipal HealthCareAuthority[
^^I^^I^^Igenerates triggerToken
^^I^^I^^Iknows private ephemeral_sk
^^I^^I^^Im1 = ENC(ephemeral_sk, triggerToken)
^^I^^I]

^^I// The brackets around m1 here mean that the value is guarded
^^I// ie: an active attacker cannot inject a value in its place

^^IHealthCareAuthority→ BackendServer : [m1]
^^IHealthCareAuthority→ SmartphoneA : m1
```

Then, Alice would have to use an AEAD cipher to encrypt `SK1A` using `ephemeral_sk` as the key and `triggerToken` as additional data and send the output to the BackendServer. Note that Alice can only obtain `triggerToken` after decrypting `m1` using `ephemeral_sk`.

[1] See §2.5 for more information regarding phases in Verifpal.

```
DP-3T: Communicating with BackendServer

^^Iprincipal SmartphoneA[
^^I^^I^^Iknows private ephemeral_sk
^^I^^I^^Im1_dec = DEC(ephemeral_sk, m1)
^^I^^I^^Im2 = AEAD_ENC(ephemeral_sk, SK1A, m1_dec)
^^I^^I]

^^ISmartphoneA → BackendServer: m2
```

The Backend Server will now have to decrypt `m1` to receive the `triggerToken` in the same way that Alice did, then attempt to decrypt `m2`. If that decryption was successful, the server would obtain `SK1A` and would be sure that the value came from Alice because it is only Alice who knows both `triggerToken` and `SK1A` at the same time as defined in the protocol.

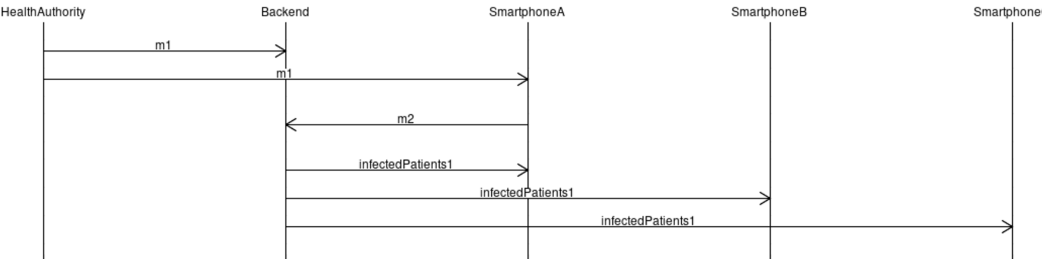
Finally, the Backend Server will add `SK1A` to the list of infected patients previously defined, and then send this list to all of the individuals in this community.

```
DP-3T: Updating List of Infected Patents

^^Iprincipal BackendServer [
^^I^^I^^Iknows private ephemeral_sk
^^I^^I^^Im2_dec = AEAD_DEC(ephemeral_sk, m2, DEC(ephemeral_sk, m1))?
^^I^^I^^IinfectedPatients1 = CONCAT(infectedPatients0, m2_dec)
^^I^^I]

^^IBackendServer → SmartphoneA: infectedPatients1
^^IBackendServer → SmartphoneB: infectedPatients1
^^IBackendServer → SmartphoneC: infectedPatients1
```

Everything that happened in Day 15 can be summarized in the following diagram:



7.3 QUERIES

Now, we may finally define the queries block, in which we ask Verifpal about the state of certain security guarantees that we expect from the protocol.

Since `SK1A` is now shared publicly, the DP-3T software running on anyone's phone should be able to re-generate all `EphID` values generated by the owner of `SK1A` starting from 14 days prior

to the day of diagnosis. These values would then be compared with the list of `EphIDs` they have received. Everyone who came in contact with Alice will therefore be notified that they have exchanged `EphIDs` with someone who has been diagnosed with the illness without revealing the identity of that person.

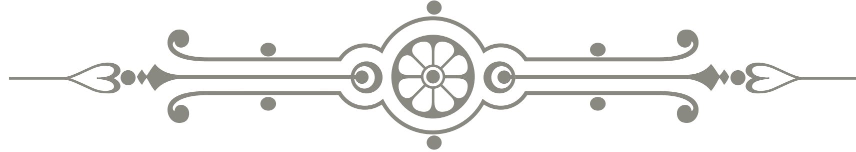
DP-3T: Queries

```
^^Iqueries[
^^I^^I^^I// Would someone who shared a value 15 days before they got tested get flagged?
^^I^^I^^I// ie in phase[0], before phase[1]
^^I^^I^^Iconfidentiality? EphID02A
^^I^^I^^I// Will people who came in contact with Alice be able to compute
^^I^^I^^I// all of Alice's EphIDs starting from Day 1
^^I^^I^^Iconfidentiality? EphID10A
^^I^^I^^Iconfidentiality? EphID11A
^^I^^I^^Iconfidentiality? EphID12A
^^I^^I^^Iconfidentiality? EphID20A
^^I^^I^^Iconfidentiality? EphID21A
^^I^^I^^Iconfidentiality? EphID22A
^^I^^I^^I// Is the server able to Authenticate Alice as the sender of m2
^^I^^I^^Iauthentication? SmartphoneA → BackendServer: m2
^^I^^I]
```

The results of our initial modeling in Verifpal suggest to us the following:

- No `EphIDs` generated by Alice are known by any parties before Alice announces her illness.
- `EphID02A` remains confidential even after Alice declaring her illness. Note that it was generated 15 days before Alice got tested.
- All of the following values `EphID10A`, `EphID11A`, `EphID12A`, `EphID20A`, `EphID21A`, `EphID22A` have been recoverable by an attacker in `phase[1]` after Alice announces her illness.

These results come in line with what is expected from the protocol. We note that the security of communication channels between Healthcare Authorities, Backend Servers, and Individuals have not been defined, and we have placed our hypothetical own security conditions with in order to focus on quickly sketching the DP-3T protocol. Further analysis will be required in order to better elucidate the extent of the obtained security guarantees.



Verifpal can guide you through an insightful and exciting investigation of the cryptographic protocols that guard the security and privacy of our daily lives.
It's up to you to decide — where will you go next?

BIBLIOGRAPHY

[1] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE Symposium on Security and Privacy (S&P)*, pages 483–502. IEEE, 2017.

[2] Vincent Cheval and Bruno Blanchet. Proving more observational equivalences with ProVerif. In *International Conference on Principles of Security and Trust*, pages 226–246. Springer, 2013.

[3] Benedikt Schmidt, Simon Meier, Cas Cremers, and David Basin. Automated analysis of Diffie-Hellman protocols and advanced security properties. In Stephen Chong, editor, *IEEE Computer Security Foundations Symposium (CSF), Cambridge, MA, USA, June 25-27, 2012*, pages 78–94. IEEE, 2012.

[4] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[5] Bruno Blanchet. CryptoVerif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar "Formal Protocol Verification Applied*, page 117, 2007.

[6] Bruno Blanchet. Security protocol verification: Symbolic and computational models. In *Proceedings of the First international conference on Principles of Security and Trust*, pages 3–29. Springer-Verlag, 2012.

[7] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On post-compromise security. In *IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 164–178. IEEE, 2016.

[8] Martín Abadi, Bruno Blanchet, and Cédric Fournet. The applied pi calculus: Mobile values, new names, and secure communication. *J. ACM*, 65(1):1:1–1:41, 2018.

[9] Ashok K Chandra and David Harel. Horn clause queries and generalizations. *The Journal of Logic Programming*, 2(1):1–15, 1985.

[10] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, et al. Verified low-level programming embedded in F. *Proceedings of the ACM on Programming Languages*, 1(ICFP):17, 2017.

[11] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. BLAKE2: simpler, smaller, fast as MD5. In *International Conference on Applied Cryptography and Network Security*, pages 119–135. Springer, 2013.

[12] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Advances in Cryptology (CRYPTO)*, pages 631–648. IACR, 2010.

[13] Colin Percival and Simon Josefsson. The scrypt password-based key derivation function. *IETF Draft URL: http://tools. ietf. org/html/josefsson-scrypt-kdf-00. txt (accessed: 30.11. 2012)*, 2016.

[14] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: new generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 292–302. IEEE, 2016.

[15] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In *Advances in Cryptology (ASIACRYPT)*, pages 552–565. Springer, 2001.

[16] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[17] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings 10th Computer Security Foundations Workshop*, pages 31–43. IEEE, 1997.

[18] Myrto Arapinis, Tom Chothia, Eike Ritter, and Mark Ryan. Defining privacy for RFID systems. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, pages 423–434. ACM, 2010.

[19] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 435–450. IEEE, 2017.

[20] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 451–466. IEEE, 2017.

[21] Carmela Tronosco et al. Decentralized privacy-preserving proximity tracing. `https://github.com/DP-3T/documents/blob/master/DP3T%20White%20Paper.pdf`, April 2020.

APPENDIX

⟨*model*⟩ ::= ⟨*attacker*⟩ ⟨*principal*⟩ (⟨*principal*⟩ | ⟨*message*⟩ | ⟨*phase*⟩)+ ⟨*queries*⟩

⟨*attacker*⟩ ::= ‘`attacker[`’ (‘`active`’ | ‘`passive`’) ‘`]`’

⟨*principal*⟩ ::= ‘`principal`’ ⟨*string*⟩ ‘`[`’ (⟨*knows*⟩ | ⟨*generates*⟩ | ⟨*leaks*⟩ | ⟨*assignment*⟩)+ ‘`]`’

⟨*knows*⟩ ::= ‘`knows `’ (‘`private`’ | ‘`public`’ | ‘`password`’) ⟨*constant*⟩ (‘`,`’ ⟨*constant*⟩)*

⟨*generates*⟩ ::= ‘`generates `’ ⟨*constant*⟩ (‘`,`’ ⟨*constant*⟩)*

⟨*leaks*⟩ ::= ‘`leaks `’ ⟨*constant*⟩ (‘`,`’ ⟨*constant*⟩)*

⟨*assignment*⟩ ::= ⟨*constant*⟩ (‘`,`’ ⟨*constant*⟩)* ‘` = `’ (⟨*primitive*⟩ | ⟨*equation*⟩)

⟨*message*⟩ ::= ⟨*string*⟩ ‘` → `’ ⟨*string*⟩ ‘`: `’ (⟨*constant*⟩ | ⟨*guardedConstant*⟩) (‘`,`’ (⟨*constant*⟩ | ⟨*guardedConstant*⟩))*

⟨*phase*⟩ ::= ‘`phase[`’ ⟨*number*⟩ ‘`]`’

⟨*queries*⟩ ::= ‘`queries[`’ (⟨*confidentialityQuery*⟩ | ⟨*authenticationQuery*⟩ | ⟨*freshnessQuery*⟩ | ⟨*unlinkabilityQuery*⟩ | ⟨*equivalenceQuery*⟩)* ‘`]`’

⟨*confidentialityQuery*⟩ ::= ‘`confidentiality? `’ ⟨*constant*⟩ ⟨*queryOptions*⟩?

⟨*authenticationQuery*⟩ ::= ‘`authentication? `’ ⟨*string*⟩ ‘` → `’ ⟨*string*⟩ ‘`: `’ ⟨*constant*⟩ ⟨*queryOptions*⟩?

⟨*freshnessQuery*⟩ ::= ‘`freshness? `’ ⟨*constant*⟩ ⟨*queryOptions*⟩?

⟨*unlinkabilityQuery*⟩ ::= ‘`unlinkability? `’ ⟨*constant*⟩ ‘`,`’ ⟨*constant*⟩ (‘`,`’ ⟨*constant*⟩)* ⟨*queryOptions*⟩?

⟨*equivalenceQuery*⟩ ::= ‘`equivalence? `’ ⟨*constant*⟩ ‘`,`’ ⟨*constant*⟩ (‘`,`’ ⟨*constant*⟩)* ⟨*queryOptions*⟩?

⟨*queryOptions*⟩ ::= ‘`[`’ ⟨*queryOption*⟩* ‘`]`’

⟨*queryOption*⟩ ::= ‘`precondition`’ ‘`[`’ ⟨*message*⟩ ‘`]`’

⟨*constant*⟩ ::= ⟨*string*⟩

⟨*guardedConstant*⟩ ::= ‘`[`’ ⟨*constant*⟩ ‘`]`’

⟨*primitive*⟩ ::= ⟨*primitiveName*⟩ ‘`(`’ (⟨*constant*⟩ | ⟨*primitive*⟩ | ⟨*equation*⟩) (‘`,`’ (⟨*constant*⟩ | ⟨*primitive*⟩ | ⟨*equation*⟩))* ‘`)`’ [‘`?`’]

⟨*equation*⟩ ::= ⟨*constant*⟩ ‘`^`’ ⟨*constant*⟩

⟨*primitiveName*⟩ ::= ‘`BLIND`’ | ‘`UNBLIND`’ | ‘`RINGSIGN`’ | ‘`RINGSIGNVERIF`’ | ‘`PW_HASH`’ | ‘`HASH`’ | ‘`HKDF`’ | ‘`AEAD_ENC`’ | ‘`AEAD_DEC`’ | ‘`ENC`’ | ‘`DEC`’ | ‘`MAC`’ | ‘`ASSERT`’ | ‘`CONCAT`’ | ‘`SPLIT`’ | ‘`SIGN`’ | ‘`SIGNVERIF`’ | ‘`PKE_ENC`’ | ‘`PKE_DEC`’ | ‘`SHAMIR_SPLIT`’ | ‘`SHAMIR_JOIN`’

Figure 1: Verifpal language syntax.

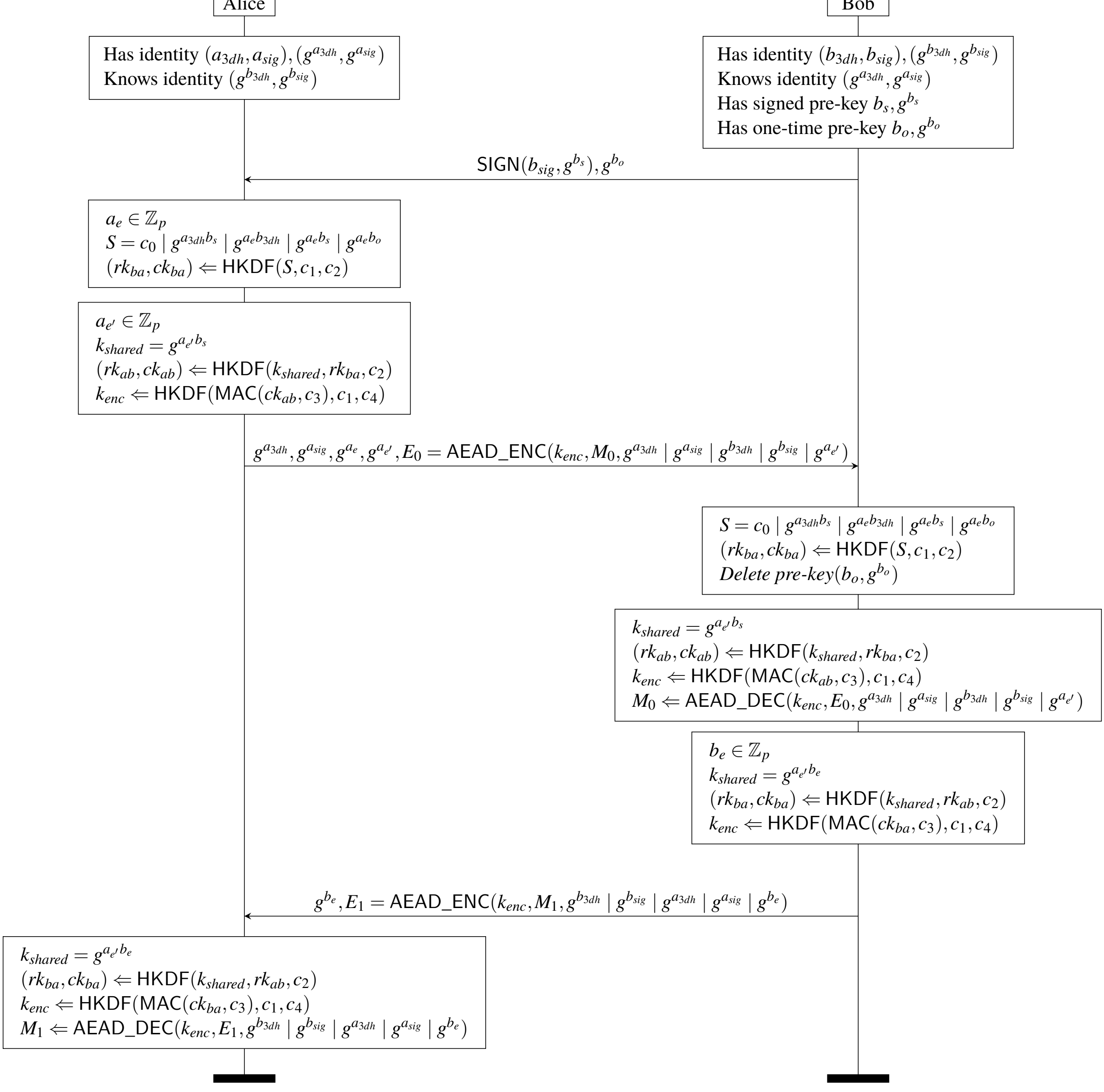


Figure 2: The Signal protocol (simplified). Alice requests a signed pre-key from Bob (via the server) and sends an initial message M_0. Bob accomplishes his side of the key exchange and obtains M_0.

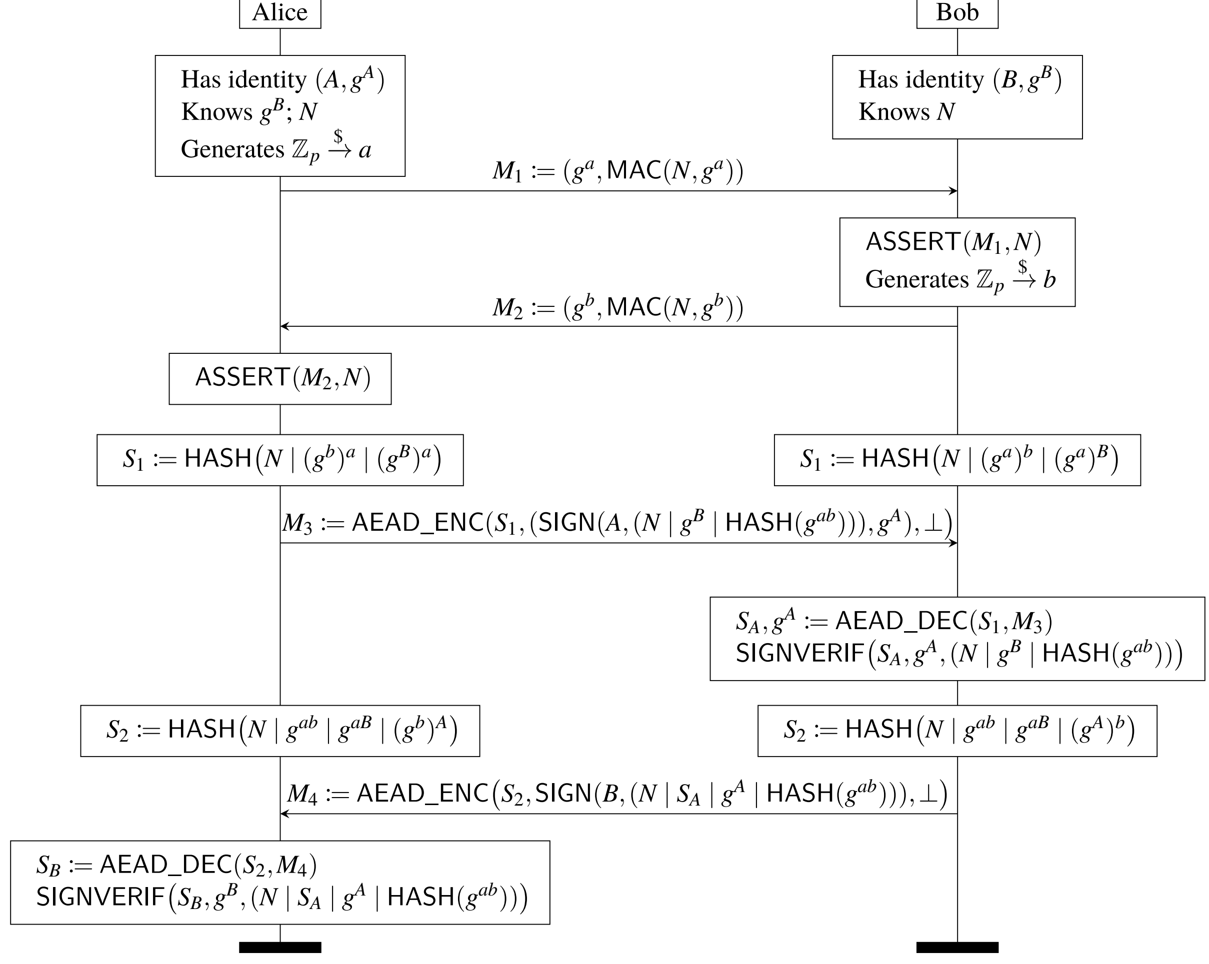


Figure 3: Secure Scuttlebutt's Authenticated Key Exchange (AKE) phase. Here, Bob acts as the server; Alice is assumed to have a pre-authenticated copy of Bob's long-term public key g^B before initializing the session. The AKE attempts to accomplish identity hiding with respect to Alice, key compromise impersonation resistance, and forward secrecy.

NOTES

This is Print 19 of the First Edition of the Verifpal User Manual.

Print 19 (April 14, 2026)

- Better specify how password extraction works in §4.5.

Print 18 (March 1, 2026)

- Updated Chapter 4 to reflect the engine redesign: replaced the multi-stage active attacker search (Stages 0–10 with equation bypass) with a bounded-depth search parameterized by a single depth value *d* (default 3), including the precise coverage guarantee.
- Replaced *before_mutate*/*before_rewrite*/*assigned* value terminology with *original*/*pre_rewrite*/*value* and explicit provenance records (*creator*, *sender*, *attacker_tainted*).
- Renamed "Genealogy of Values" to "Provenance of Values" with updated description of provenance-based authentication checking.
- Updated "Preventing State Space Explosion" section to describe depth-based budget constraints instead of stage-based constraints.
- Updated "Practical Significance" subsection to reference depth-based search instead of staged search heuristics.
- Corrected the deduction loop description: the loop is a pure fixed-point computation that does not check queries; query evaluation is a separate phase.
- Clarified the three-phase pipeline (trace generation, knowledge closure, query evaluation) and that these phases are never interleaved.
- Updated Chapter 3 to reference "provenance" instead of "genealogy" and "bounded-depth search" instead of "checking queries after each deduction".
- Expanded the equational theory listing in Chapter 2 with decomposition rules (`ENC`, `BLIND`), passive reveal rules (`AEAD_ENC`, `CONCAT`), the `RINGSIGNVERIF` rewrite rule, and rule-type annotations.

- Fixed Scuttlebutt model listing: corrected `longTermAPub_bob` to `longTermAPub_Bob` (case mismatch).
- Updated Verifpal version requirement for VS Code extension from 0.13.0 to 0.40.0.
- Removed stale `make all` cross-compilation reference; replaced with current `make build/test/lint` targets.

Print 17 (February 24, 2026)

- Radically expanded Chapter 4 (Analysis in Verifpal) with precise descriptions of the verification pipeline, the three-phase deductive analysis loop, the staged active attacker search with budget parameters, skeleton-based injection filtering, guard bypass, and value resolution semantics.
- Added new sections on soundness and completeness (§4.6) with precise formal claims: attack soundness (reported attacks are genuine), deduction fixed-point completeness, and explicit statements of what Verifpal does not guarantee (verification completeness, unbounded sessions, computational soundness).
- Added equational theory section (§2.3.4) to Chapter 2 listing the complete set of rewrite equations governing Verifpal's symbolic model.
- Improved formal precision of all five query type descriptions in Chapter 3: confidentiality, authentication, freshness, unlinkability, and equivalence queries now include formal definitions of when each query is contradicted.
- Expanded the Dolev-Yao model footnote in the Introduction with explanation of what the symbolic model means in practice.
- Clarified the asymmetry between attack-finding (sound) and verification (not complete) in the Introduction.
- Corrected `SIGNVERIF` and `RINGSIGNVERIF` output descriptions from `message` to `verified` in Chapter 2.
- Improved PrimitiveSpec rule descriptions with clarification of the distinction between Rebuild (symbolic evaluation) and Recompose (attacker analysis).
- Improved checked primitives description with precise explanation of truncation semantics under active attacker.
- Expanded Diffie-Hellman equation description with explanation of internal commutative multiset representation.
- Added ring signature citation [15].
- Improved passive and active attacker descriptions in Chapter 3 with precise deduction loop characterization.
- Updated VerifHub references to Verifpal Workbench.
- Updated decomposition rules list to include `PKE_ENC`, `PKE_DEC`, and `BLIND`; updated rewrite rules list to include `RINGSIGNVERIF` and `UNBLIND`.

Print 16 (February 11, 2026)

- Verifpal is now written in Rust: updated build-from-source instructions to use `cargo build` instead of Go and `make`.
- Switched manual build system from LuaLaTeX to Tectonic.
- Removed documentation for the `translate` command.
- Added explicit Dolev-Yao model framing throughout, with new citation [4].
- Added caveat in Chapters and 4 clarifying that Verifpal is a protocol analysis tool, not a proof-producing verifier.
- Toned down soundness claims in Chapter 4: Verifpal now described as a bounded analysis tool with heuristics.
- Added reference to Lowe's hierarchy of authentication specifications [17] in Chapter 3.
- Formalized freshness query definition in Chapter 3.
- Added citation for unlinkability [18] in Chapter 3.
- Expanded equivalence query description in Chapter 3.
- Corrected `BLIND` primitive output label from `m` to `blinded`.
- Rewrote phases semantics as a structured list in Chapter 2.
- Replaced non-standard term "future secrecy" with "post-compromise security" and rewrote its definition.
- Rewrote one-time pre-key signing explanation in Chapter 5.
- Various corrections and formatting improvements.

Print 15 (May 7, 2021)

- Add new section in Chapter 3 documenting equivalence queries introduced in Verifpal 0.23.0.
- Added a *Are You Sure It's Private?* information box.

Print 14 (October 25, 2020)

- Some clarifications to Verifpal's analysis goals in Chapter 4.

Print 13 (June 28, 2020)

- Add a small section about VerifHub to Chapter 1.

Print 12 (April 30, 2020)

- Add new primitives: `BLIND` and `UNBLIND`.
- Expand documentation for Verifpal for Visual Studio Code.

Print 11 (April 15, 2020)

- Add new sections in Chapter 3 documenting freshness and unlinkability queries as introduced in Verifpal 0.12.0.
- Add freshness and unlinkability queries to the Verifpal syntax table.
- Correct typos.

Print 10 (April 11, 2020)

- Add a new chapter on the DP-3T protocol, contributed by Georgio Nicolas.

Print 9 (February 29, 2020)

- Introduce the notion of *"core primitives"*. Make `ASSERT` a core primitive.
- Add new core primitives: `CONCAT` and `SPLIT`.

Print 8 (February 6, 2020)

- Rewrite Signal forward secrecy section.
- Add documentation for `leaks` declaration.
- Add documentation for query options.

Print 7 (January 29, 2020)

- Add documentation for phases.

Print 6 (January 25, 2020)

- Add new primitive: `PW_HASH`.
- Add documentation for the `password` qualifier for `knows` declarations.

Print 5 (January 10, 2020)

- Add new primitives: `PKE_ENC`, `PKE_DEC`, `SHAMIR_SPLIT`, `SHAMIR_JOIN`.
- Add installation instructions for the Homebrew package manager.
- Minor corrections and changes.

Print 4 (September 9, 2019)

- Rename `HMACVERIF` to `ASSERT` and `HMAC` to `MAC`.

Print 3 (September 3, 2019)

- Reformatted book in preparation for hardcover textbook printing.
- Some minor changes and additions.

Print 2 (August 31, 2019)

- Added instructions for building Verifpal from source on Windows.
- Fixed some inaccuracies in Chapter 6.
- Fixed some grammar errors.

Print 1 (August 26, 2019)

- Initial Print.