# Verifpal
# User Manual

FIRST EDITION

Nadim Kobeissi

Published by Symbolic Software

Copyright © 2024 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 November 11, 2024*

# *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.*

# 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 goals: 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 goals *confidentiality* and *authentication*, respectively.

It's important to be able to verify whether TLS and Signal actually accomplish these goals. 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 goals it's supposed to accomplish[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.

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.

---

[2] ProVerif, Tamarin and also Verifpal verify protocols in the *symbolic model*. Computational protocol verifiers, such as CryptoVerif [4] also exist, but the computational model and its differences compared to the symbolic model [5] are not within the purview of this manual.

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* [6]. 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 the same question about the future, roughly speaking.

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"* [7]). It also expects the user to intuitively reason about protocols as Horn clauses [8] 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 [9]. 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.

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.

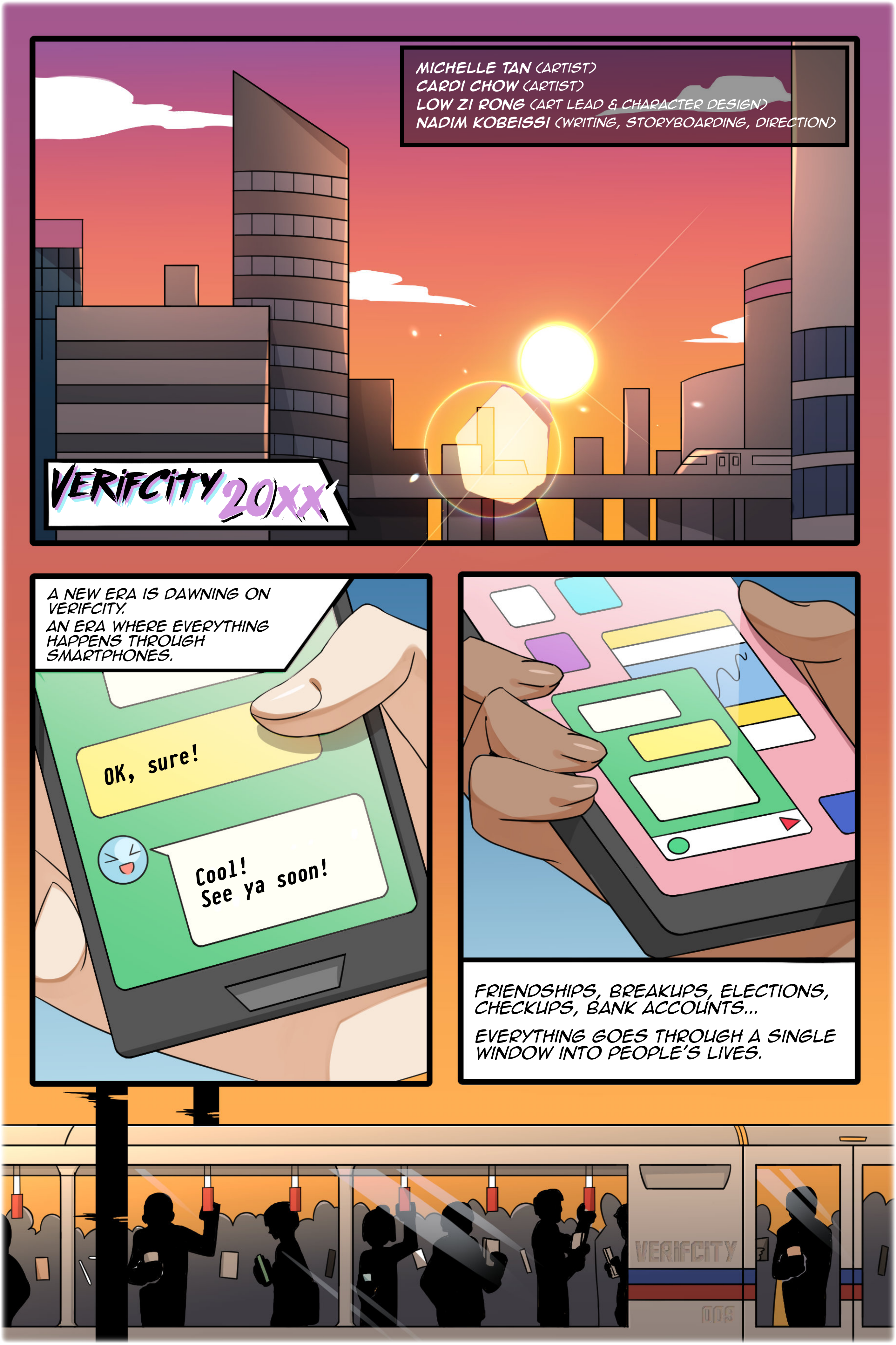
Like all heroes, Verifpal thrives in the midst of adventure. What protocols will you and Verifpal venture within? What interesting discoveries will you make?

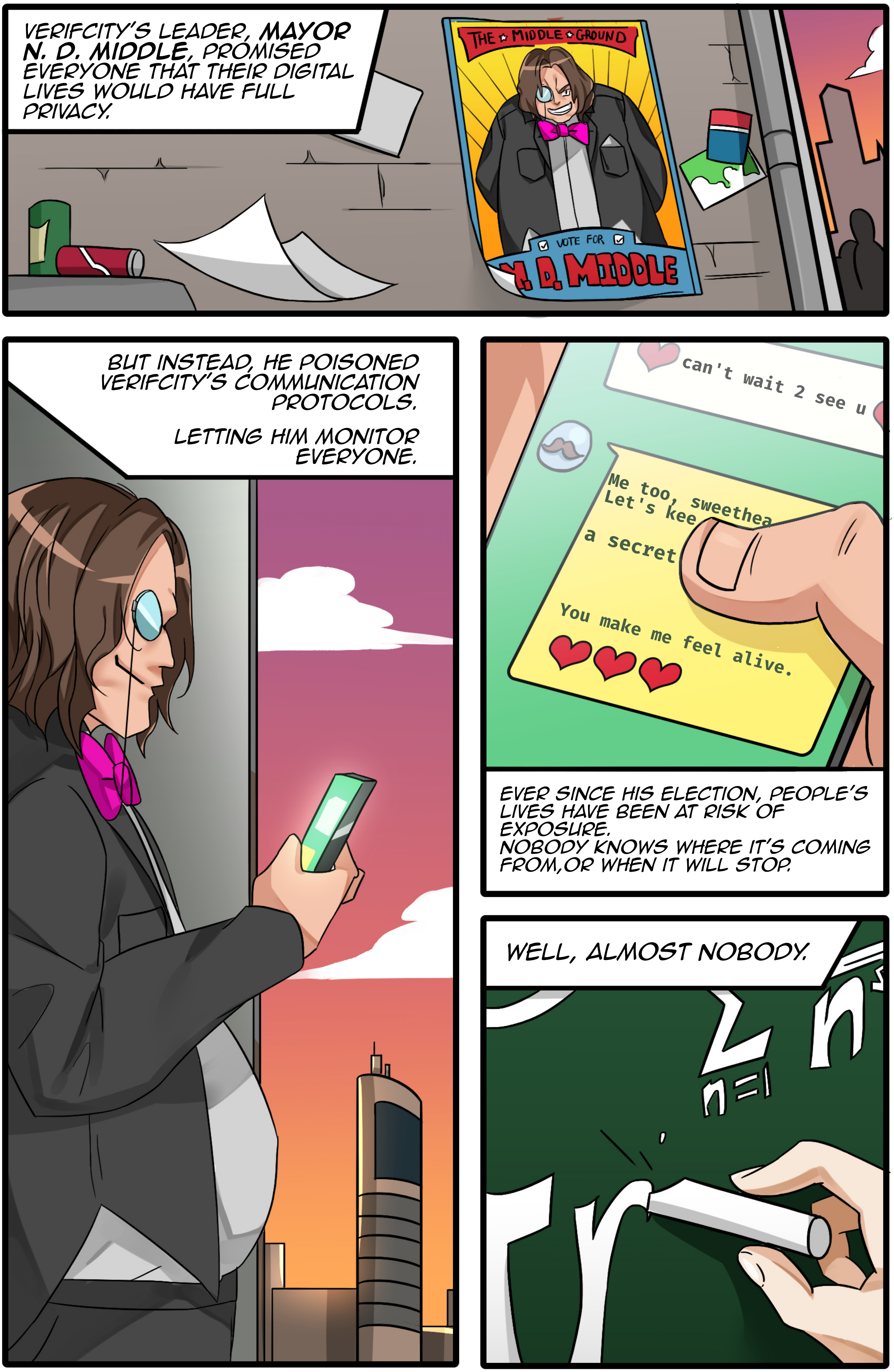
*Nadim Kobeissi*

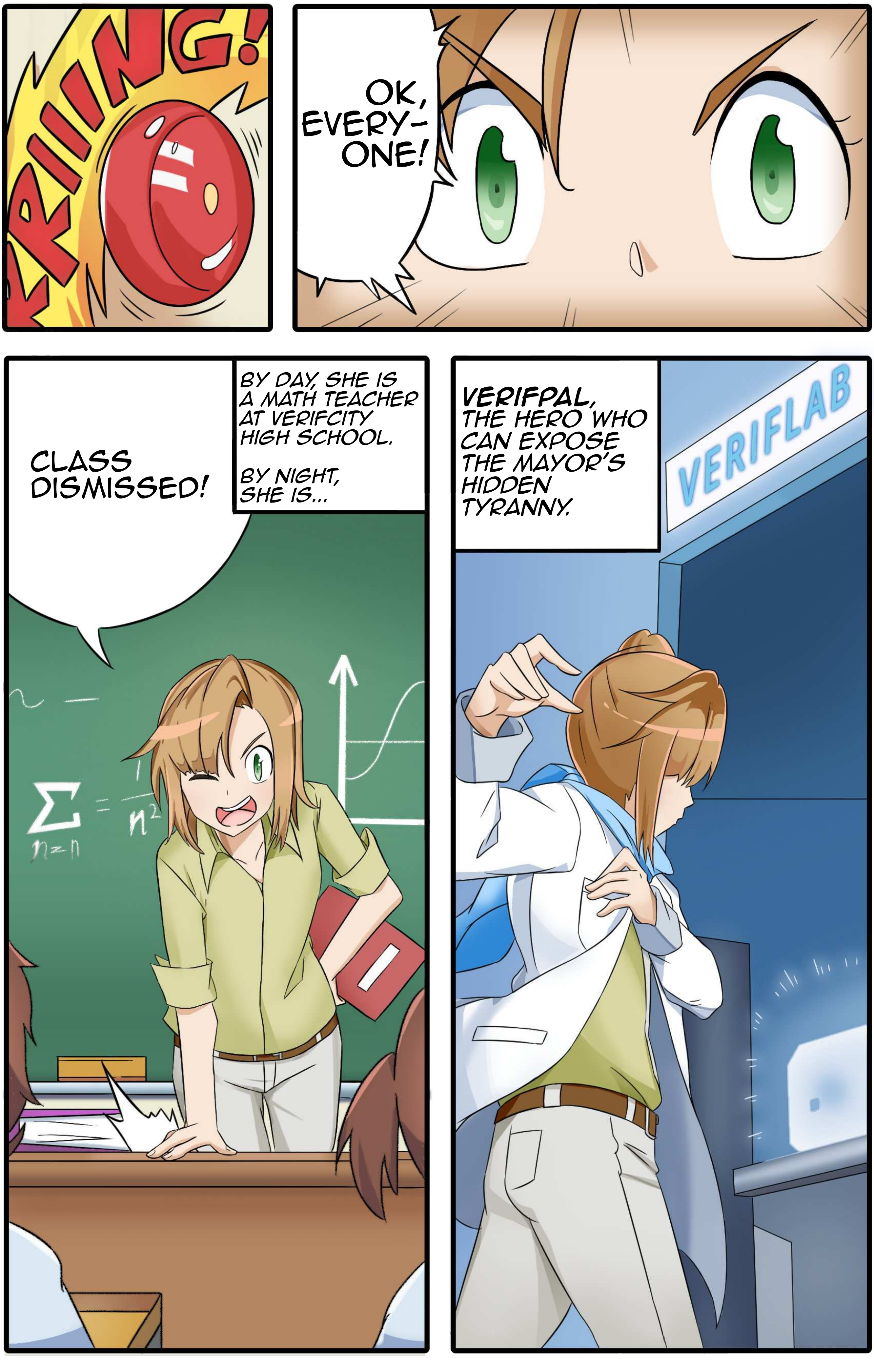
*July 27, 2019*

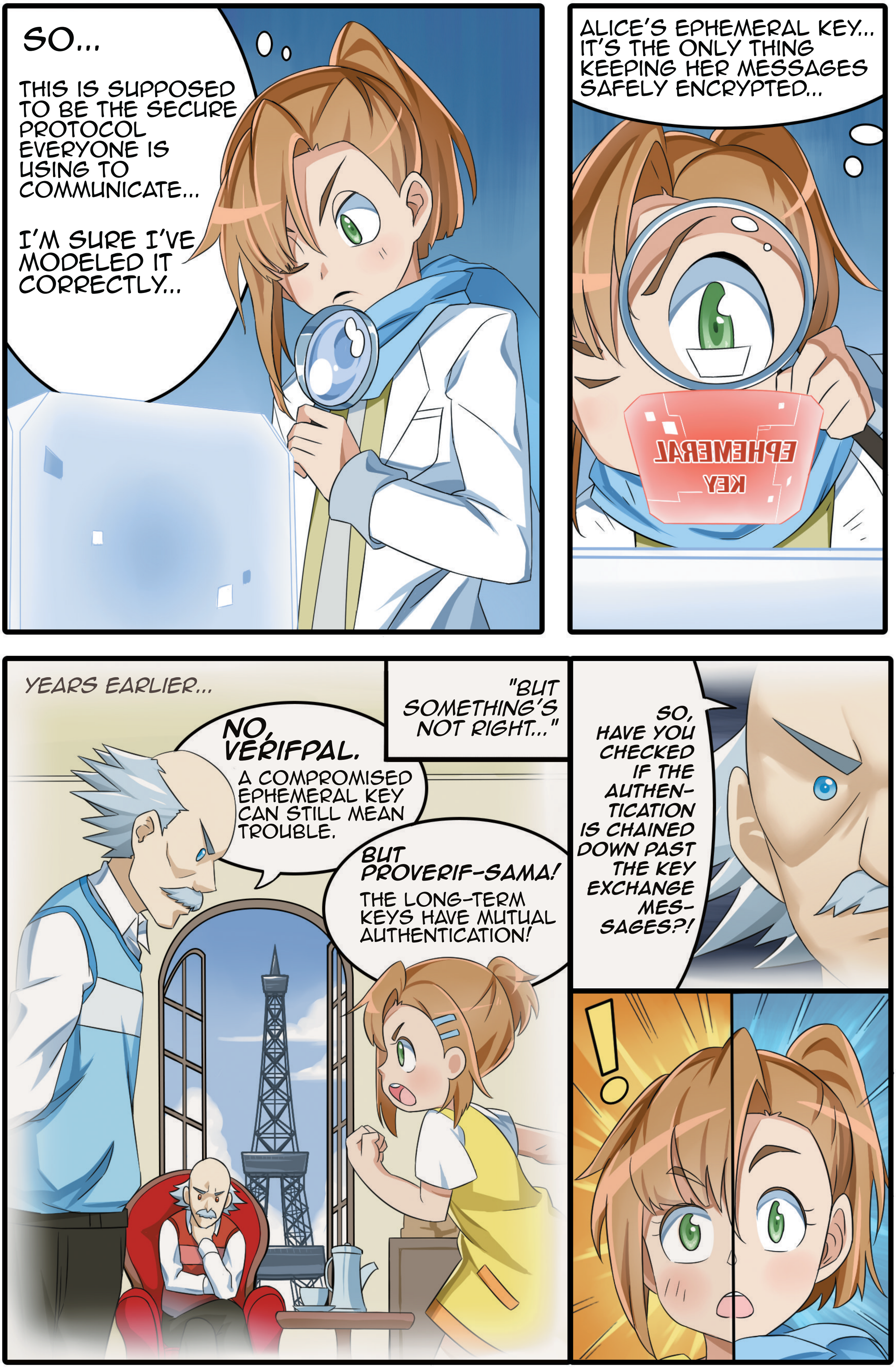
---

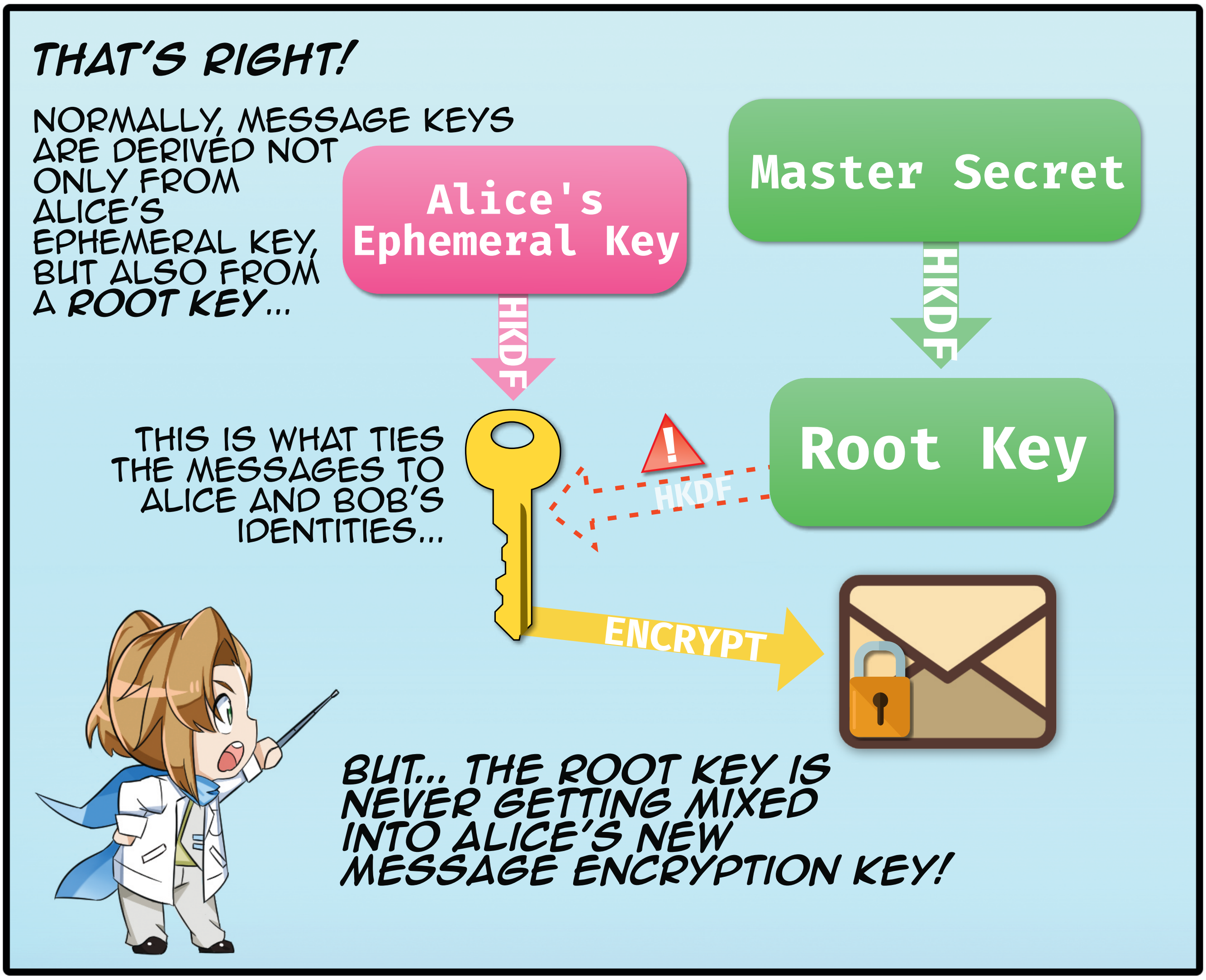
[3] This is an example of how Verifpal fundamentally diverges from ProVerif when it comes to certain goals — its focus on ease of use will allow ProVerif, for the foreseeable future, to provide more elaborate models due to, for example, support for user-defined primitives.

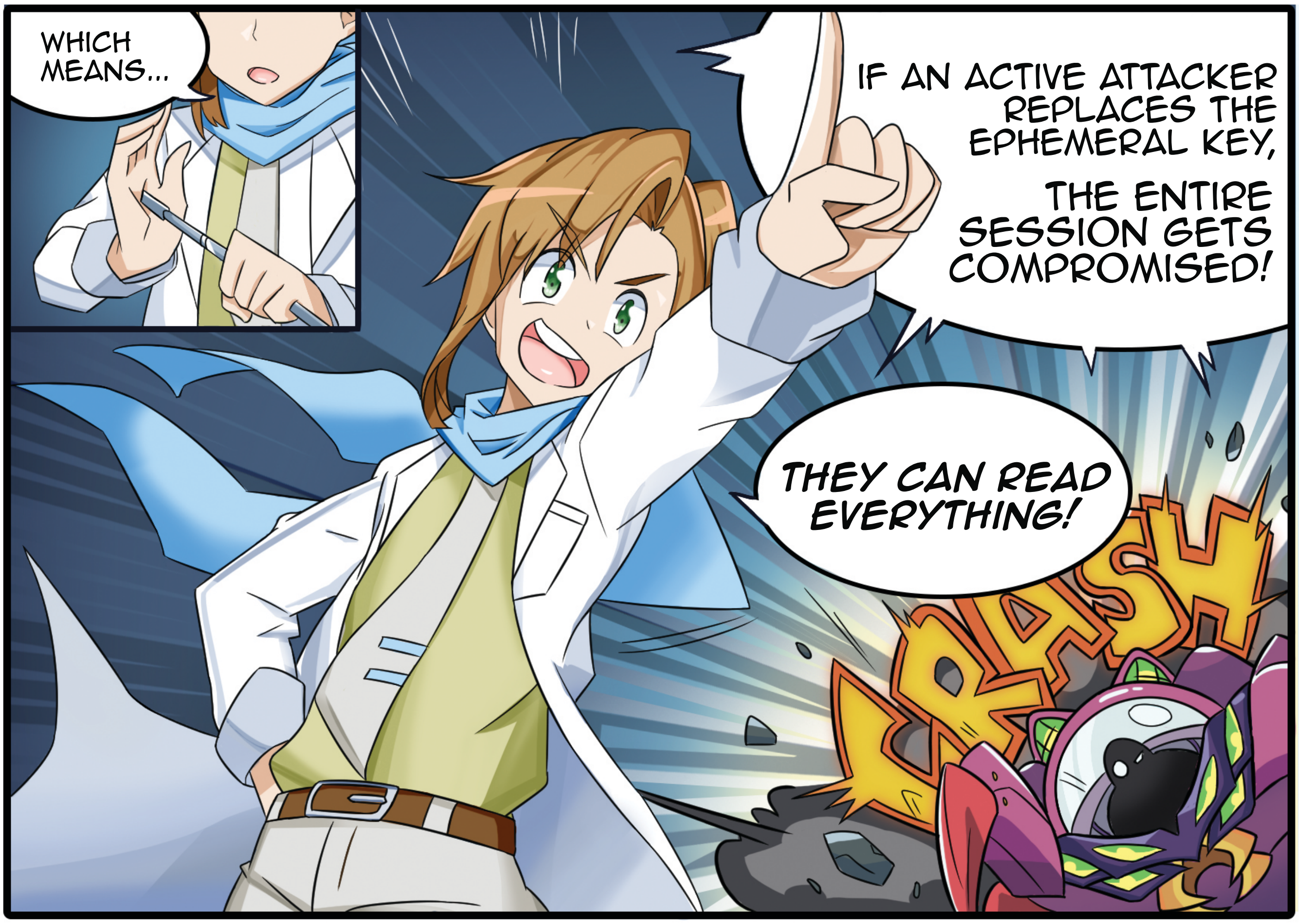

MICHELLE TAN (ARTIST)
CARDI CHOW (ARTIST)
LOW ZI RONG (ART LEAD & CHARACTER DESIGN)
NADIM KOBEISSI (WRITING, STORYBOARDING, DIRECTION)
VERIFCITY 20XX
A NEW ERA IS DAWNING ON VERIFCITY. AN ERA WHERE EVERYTHING HAPPENS THROUGH SMARTPHONES.
OK, sure!
Cool! See ya soon!
FRIENDSHIPS, BREAKUPS, ELECTIONS, CHECKUPS, BANK ACCOUNTS...
EVERYTHING GOES THROUGH A SINGLE WINDOW INTO PEOPLE'S LIVES.
VERIFCITY

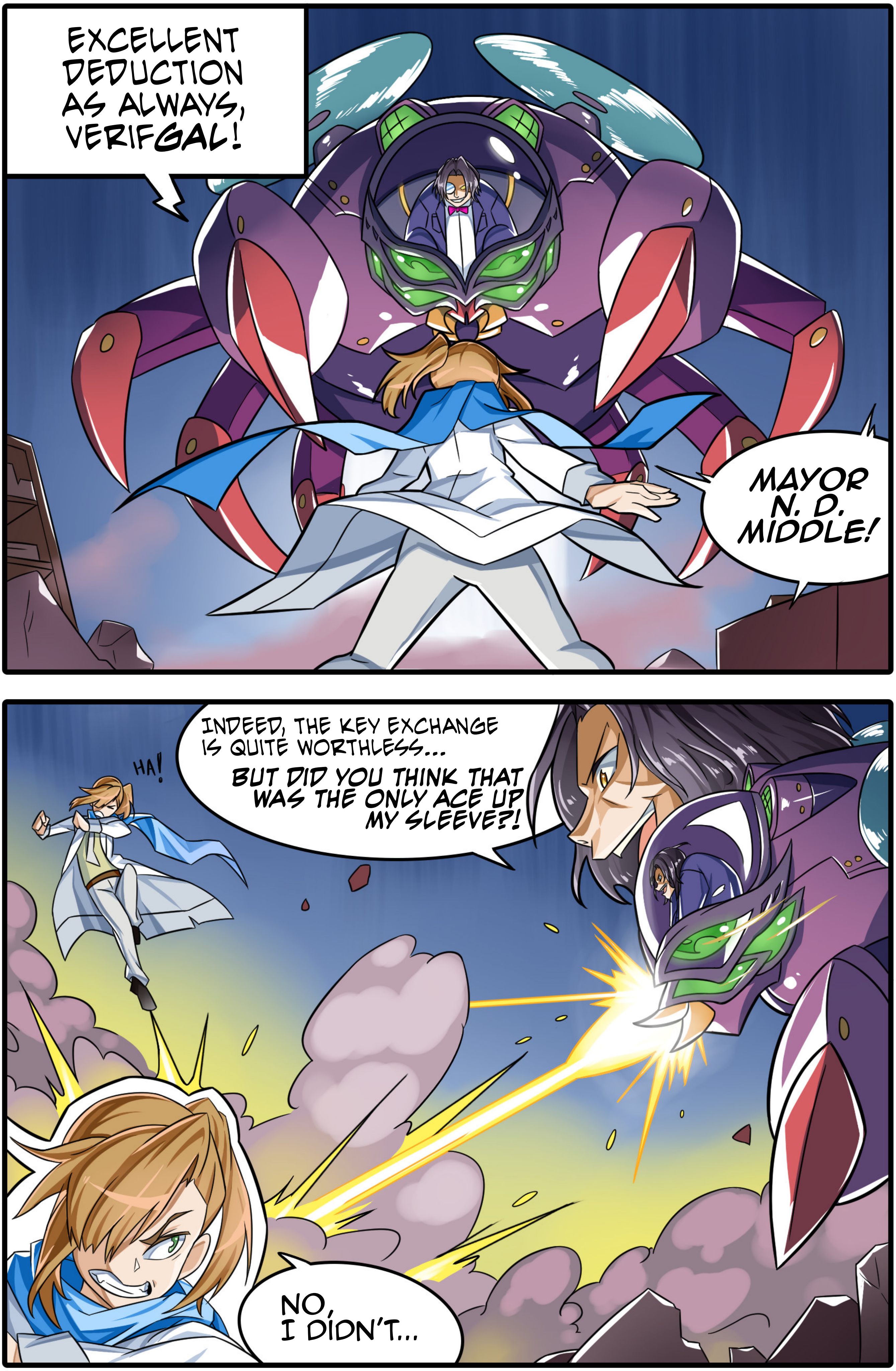

VERIFCITY'S LEADER, **MAYOR N. D. MIDDLE**, PROMISED EVERYONE THAT THEIR DIGITAL LIVES WOULD HAVE FULL PRIVACY.
THE ⭑ MIDDLE ⭑ GROUND
☑ VOTE FOR ☑
N. D. MIDDLE
BUT INSTEAD, HE POISONED VERIFCITY'S COMMUNICATION PROTOCOLS.
LETTING HIM MONITOR EVERYONE.
can't wait 2 see u
Me too, sweethea
Let's kee
a secret
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.
n=1

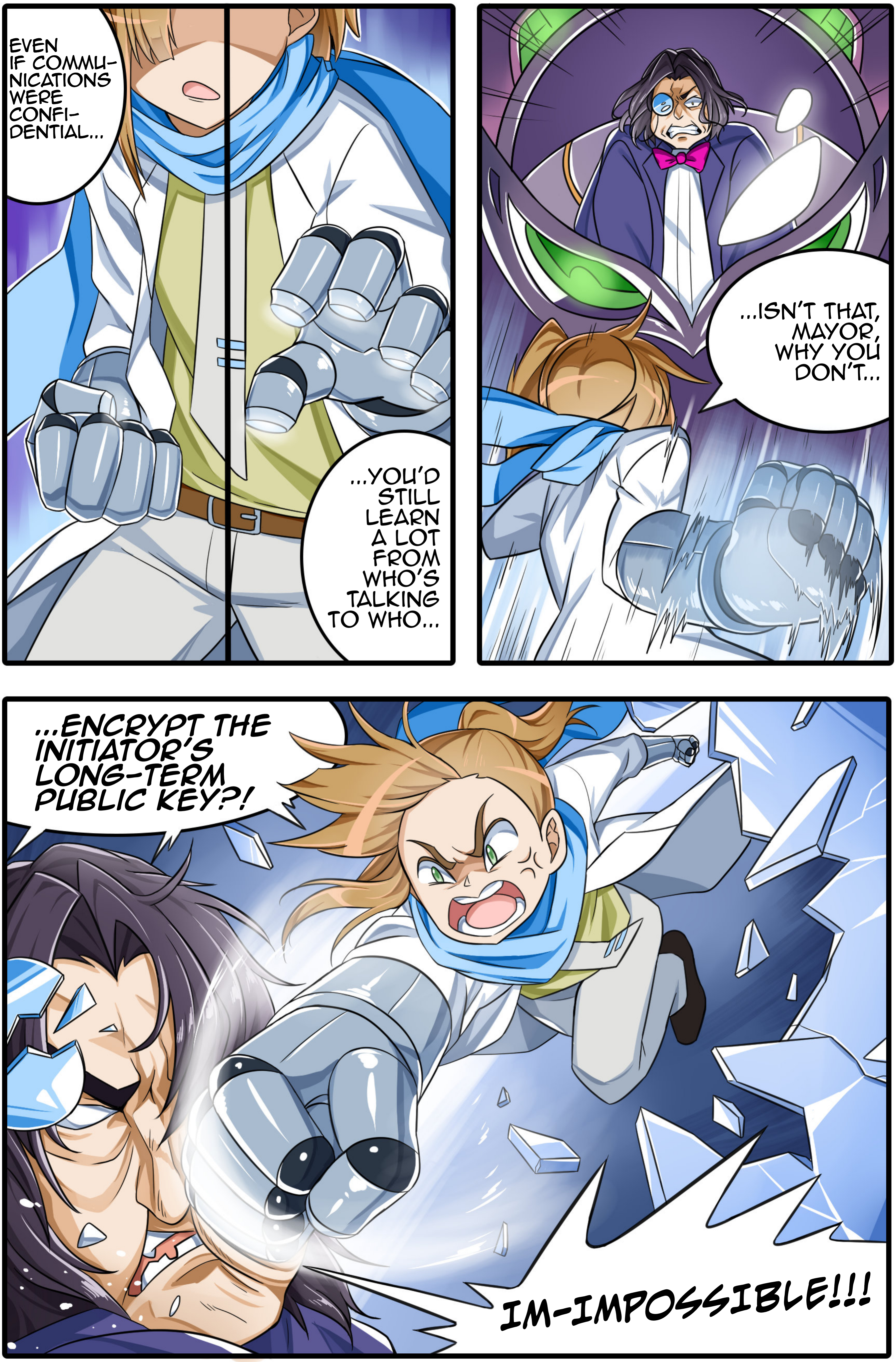

RIIING!
OK, EVERY-ONE!
CLASS DISMISSED!
BY DAY, SHE IS A MATH TEACHER AT VERIFCITY HIGH SCHOOL.
BY NIGHT, SHE IS...
VERIFPAL, THE HERO WHO CAN EXPOSE THE MAYOR'S HIDDEN TYRANNY.
VERIFLAB

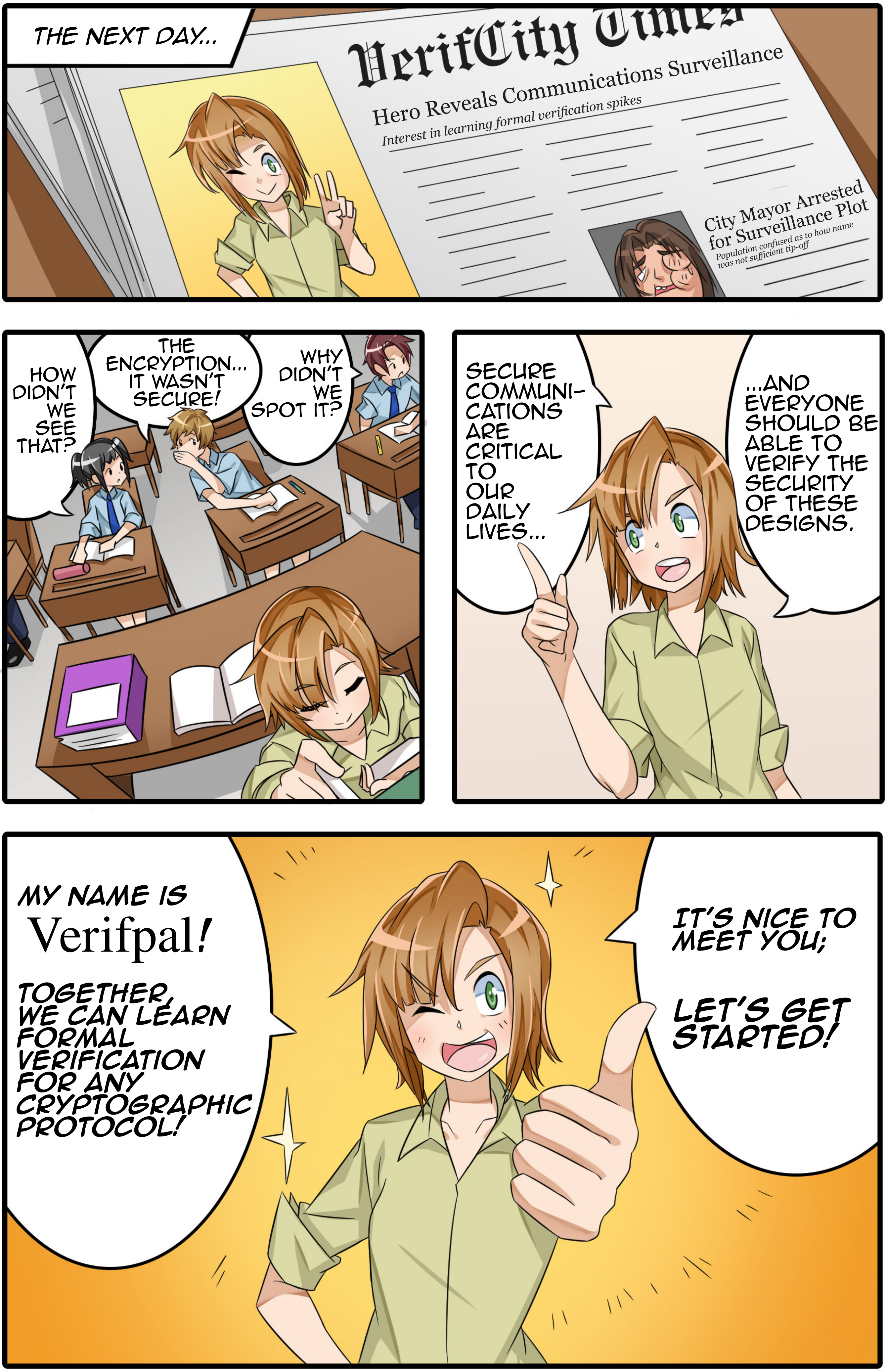

SO...
THIS IS SUPPOSED TO BE THE SECURE PROTOCOL EVERYONE IS USING TO COMMUNICATE...
I'M SURE I'VE MODELED IT CORRECTLY...
ALICE'S EPHEMERAL KEY... IT'S THE ONLY THING KEEPING HER MESSAGES SAFELY ENCRYPTED...
EPHEMERAL KEY
YEARS EARLIER...
"BUT SOMETHING'S NOT RIGHT..."
NO, VERIFPAL.
A COMPROMISED EPHEMERAL KEY CAN STILL MEAN TROUBLE.
BUT PROVERIF-SAMA!
THE LONG-TERM KEYS HAVE MUTUAL AUTHENTICATION!
SO, HAVE YOU CHECKED IF THE AUTHENTICATION IS CHAINED DOWN PAST THE KEY EXCHANGE MESSAGES?!
!


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...


EVEN IF COMMUNICATIONS WERE CONFIDENTIAL...
...YOU'D STILL LEARN A LOT FROM WHO'S TALKING TO WHO...
...ISN'T THAT, MAYOR, WHY YOU DON'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 . . . . . . . . . . . . . . . . . . . . . . . 5

1.7 Sharing Verifpal Models with VerifHub . . . . . . . . . . . . . . . . . . . . 6

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 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

2.5 Phases . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.6 Queries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.7 Comments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
2.8 A Simple Complete Example . . . . . . . . . . . . . . . . . . . . . . . . 16

**3 Protocols and Queries in Verifpal 17**

3.1 Use Cases and Security Goals . . . . . . . . . . . . . . . . . . . . . . . 18
3.2 Queries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
3.3 Passive and Active Attackers . . . . . . . . . . . . . . . . . . . . . . . 24
3.4 Understanding Verification Results . . . . . . . . . . . . . . . . . . . . 26
3.5 Modeling a Challenge-Response Protocol . . . . . . . . . . . . . . . . . . 28

**4 Analysis in Verifpal 29**

4.1 Analysis Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.2 Preventing State Space Explosion . . . . . . . . . . . . . . . . . . . . . 31
4.3 Soundness of Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

**II Protocol Examples in Verifpal 35**

**5 Secure Messaging with Signal 36**

5.1 Security Goals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
5.2 Principals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
5.3 Queries and Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

**6 Gossip with Scuttlebutt 43**

6.1 Security Goals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
6.2 Principals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
6.3 Queries and Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

**7 Contact Tracing with DP-3T** **49**

7.1 Security Goals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

7.2 Modeling DP-3T . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

7.3 Queries . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

**Bibliography** **57**

**Appendix** **59**

**Notes** **63**

# 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`.

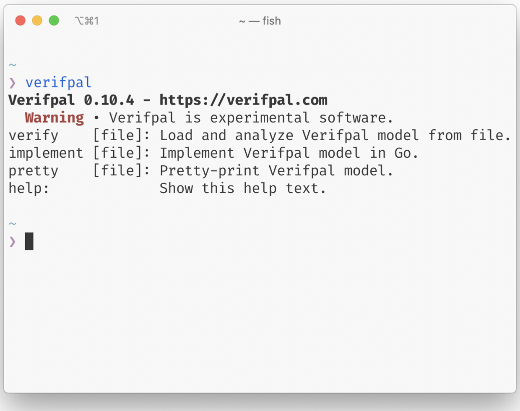
### *Verifpal is Beta Software*

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.

---

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

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



```
~
❯ 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.
- `translate`: generates a ProVerif[3] or Coq[4] model based on the given Verifpal model, which

---

[3] `https://proverif.inria.fr`.
[4] `https://coq.inria.fr`.

may be useful for further analysis.

- `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 that this is Verifpal version `0.1`. 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*[5] instructions in order to understand how to best install Git for your computer and operating system.

**Installing Go.** You must have the Go programming language installed in order to build Verifpal. Please Review the *Go Getting Started*[6] instructions in order to understand how to best install Go for your computer and operating system.

**Installing Dependencies.** Verifpal relies on the Pigeon PEG parser generator and on the Aurora ANSI color printer as dependencies. Once you have installed Git and cloned the Verifpal repository, simply type `make dep` in order to install Go dependencies required by Verifpal.

**Compiling Verifpal.** On Windows, simply type `Build` (or run the included `Build.cmd` script) to build Verifpal for Windows, Linux, macOS and FreeBSD. This will also install dependencies.

---

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

[6] `https://golang.org/doc/install`

On Linux, macOS and FreeBSD, simply type `make all` instead. Binaries will then be available under the `build/` folder.

## 1.6 VERIFPAL FOR VISUAL STUDIO CODE

A Verifpal extension is also available for Visual Studio Code[7].

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:

---

[7] 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 0.13.0 or higher to be installed for all functionality to work correctly.

- `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`)[8]

## 1.7 SHARING VERIFPAL MODELS WITH VERIFHUB

VerifHub is a web-based service that allows Verifpal users to easily share and discuss Verifpal models of cryptographic protocols. The VerifHub service provides unique URIs for each shared model which includes a syntax-highlighted model, an automatically generated diagram and a summary of the analysis results. Users may then share models and also discuss them using the built-in VerifHub model discussion feature.

For more information on how to use VerifHub, please visit `https://verifhub.verifpal.com`.

## 1.8 NEWS AND DISCUSSION

Sign up to the Verifpal Mailing List[9] in order to stay informed on the latest news and announcements regarding Verifpal, and to participate in Verifpal-related discussions!

For more informal or immediate Verifpal discussion, please join the Verifpal Discord channel.[10]

---

[8] If you have installed Verifpal via the Snap Store, you will need to set `verifpal.path` to `/snap/verifpal/current/bin/verifpal`.

[9] `https://verifpal.com/list`

[10] `https://verifpal.com/discord`

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

```
principal Alice[
  knows public c0, c1
  knows private m1
  generates a
]
```

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

New Principal: Bob

```
principal Bob[
  knows public c0, c1
  knows private m2
  generates b
  gb = G^b
]
```

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 essentially *"perfect"*. That is to say, hash functions are perfect one way functions, and not susceptible to something like length extension attacks. It is also not possible to model for, say, encryption primitives that use 40-bit keys, which could be guessed easily, since encryption functions are perfect pseudo-random permutations, and so on[1].

Internally in Verifpal's standard implementation, all primitives are defined using a common spec called PrimitiveSpec which restricts how they can be expressed to a set of common rules. Aside

---

[1] That is, incidentally, the fundamental difference between tools like Verifpal, ProVerif and Tamarin, which operate in the *symbolic* model, and software like CryptoVerif [4] which operates in the computational model. The computational model allows CryptoVerif to capture ideas such as key length, which can make for more accurate modeling in some instances.

Simple Protocol

```
attacker[active]
principal Alice[
  generates a
  ga = G^a
]
Alice → Bob: ga
principal Bob[
  knows private m1
  generates b
  gb = G^b
  ss_a = ga^b
  e1 = AEAD_ENC(ss_a, m1, gb)
]
Bob → Alice: gb, e1
principal Alice[
  ss_b = gb^a
  e1_dec = AEAD_DEC(ss_b, e1, gb)?
]
```

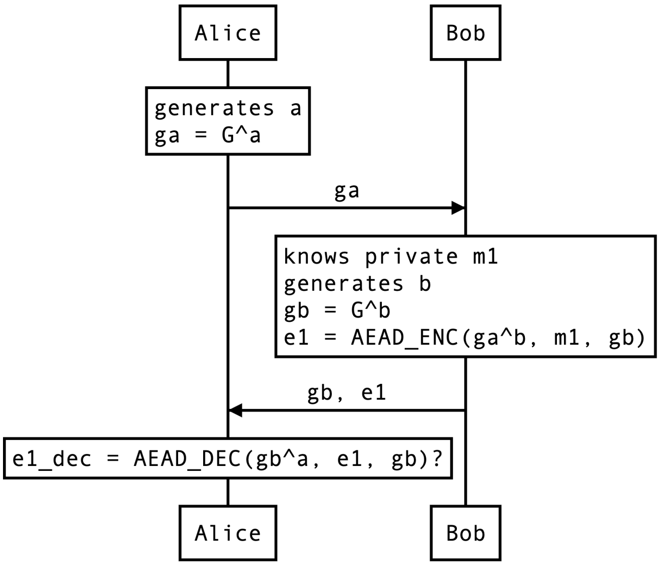


**Figure 2.1:** A complete example model of a simple protocol is shown on the left. On the right, a helpful diagram is provided to illustrate how modeling in Verifpal works. The diagram on the right is not part of Verifpal's modeling language and is simply provided here as a visual aid.

from information such as the primitive's names, arity and number of outputs, each PrimitiveSpec defines a primitive 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 inputs. (Given `ENC(k, m)` and `k`, reveal `m`).
- Recompose. Given a defined subset of a primitive's outputs, reveal one of its inputs. (Given `a, b`, reveal `x` if `a, b, _ = SHAMIR_SPLIT(x)`).
- Rewrite. Given a matching defined pattern within a primitive's inputs, rewrite the primitive expression itself into a logical subset of its inputs. (Given `DEC(k, ENC(k, m))`, rewrite the entire expression `DEC(k, ENC(k, m))` to `m`).
- Rebuild. Given a primitive whose inputs are all the outputs of some same other primitive, rewrite the primitive expression itself into a logical subset of its inputs. (Given `SHAMIR_JOIN(a, b)` where `a, b, c = SHAMIR_SPLIT(x)`, rewrite the entire expression `SHAMIR_JOIN(a, b)` to `x`).

### 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 [10].
  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 [11].
  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 [12] or Argon2 [13]. 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`.

### 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.

---

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

- `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`.
  Classic signature primitive. Here, `key` is a private key, for example `a`.

- `SIGNVERIF(G^key, message, SIGN(key, message)): message`.
  Verifies if signature can be authenticated.
  If key `a` was used for `SIGN`, then `SIGNVERIF` will expect `G^a` as the key value. Output value is not necessarily used; see §2.3.2 below for information on how to validate this check.

- `RINGSIGN(key_a, G^key_b, G^key_c, message): signature`.
  Ring signature.
  In ring signatures, one of three parties (Alice, Bob and Charlie) signs a message. The resulting signature can be verified using the public key of any of the three parties, and the signature does not reveal the signatory, only that they are a member of the signing ring (Alice, Bob or Charlie). The first key must be the private key of the actual signer, while the subsequent two keys must be the public keys of the other potential signers.

- `RINGSIGNVERIF(G^a, G^b, G^c, m, RINGSIGN(a, G^b, G^c, m)): m`.
  Verifies if a ring signature can be authenticated.
  The signer's public key must match one or more of the public keys provided, but the public keys may be provided in any order and not necessarily in the order used during the `RINGSIGN` operation. Output value is not necessarily used; see §2.3.2 below for information on how to validate this check.

- `BLIND(k, m): m`.
  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 [14], 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 model execution will abort should `AEAD_DEC` fail authenticated decryption, or 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.

If you are analyzing under a passive attacker, then Verifpal will only execute the model once. Therefore, if a checked primitive fails, the entire verification procedure will abort. Under an active attacker, however, Verifpal is forced to execute the model once over for every possible permutation of the inputs that can be affected by the attacker. Therefore, a failed checked primitive may not abort all executions — and don't forget, messages obtained before the failure of the checked primitive are still valid for analysis, perhaps even in future sessions. 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:

### *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!

Example Equations

```
principal Server[
  generates x
  generates y
  gx = G^x
  gy = G^y
  gxy = gx^y
  gyx = gy^x
]
```

In the above, `gxy` and `gyx` are considered equivalent by Verifpal. In Verifpal, all equations must have the constant `G` as their root generator. This mirrors Diffie-Hellman behavior. Furthermore, all equations can only have two constants (`a^b`), but as we can see above, equations can be built on top of other equations (as in the case of `gxy` and `gyx`).

## 2.4 MESSAGES

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

Example: Messages

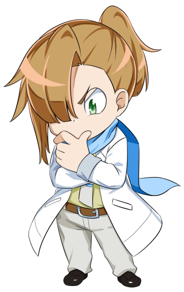
```
Alice → Bob: ga, e1
Bob → Alice: [gb], e2
```

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.

---

[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.



### *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?

## 2.5 PHASES

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

Example: Phases

```
principal Alice[...]
principal Bob [...]
Bob → Alice: b1

phase[1]

principal Alice[leaks a2]
```

In the above example, the attacker won't be able to learn `a2` until the execution of everything that occurred in phase 0 (the initial phase of any model) is concluded. Furthermore, the attacker can only manipulate `a2` within the confines of the phases in which it is communicated. That is to say, the attacker will have knowledge of `b1` when doing analysis in phase 1, but won't be able to manipulate `b1` in phase 1. The attacker won't have knowledge of `a2` during phase 0, but will be able to manipulate `b1` in phase 0.

Values are learned at the earliest phase in which they are communicated, and can only be manipulated within phases in which they are communicated, which can be more than one phase since Alice can for example send a2 later to Carol, to Damian, etc. Importantly, 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

```
queries[
  confidentiality? e1
  confidentiality? m1
  authentication? Bob -> Alice: e1
  equivalence? ss_a, ss_b
]
```

Under a passive attacker, the answers would be *“yes”*, *“no”* and *“no”*. Under an active attacker, the answer to all three questions would be *“yes”*. 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

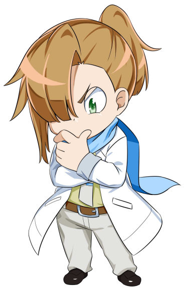
So far, this manual has assumed that you have an understanding of the kind of thinking that determines the design of a cryptographic protocol: the use cases, the security goals, the principals involved. In this chapter, we will go through these concepts again, as they are central to a complete understanding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

```
queries[
  confidentiality? m1
  authentication? Alice -> Bob: e1
]
```

**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. If the answer is yes, then the attacker was able to obtain the message, despite it being presumably encrypted.

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 the 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 decrypts and authenticates `e1`, does that necessarily mean that Alice sent `e1` to Bob?"* The implication is that if the attacker was able to successfully convince Bob to validate the decryption of `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,

Example Freshness Query

```
attacker[active]
principal Alice[
  knows private a
  generates b
  ha = HASH(a)
  hb = HASH(b)
]
Alice→ Bob: ha, hb
principal Bob[
  knows private a
  _ = ASSERT(ha, HASH(a))
]
queries[
  freshness? ha
  freshness? hb
]
```

**Figure 3.2:** Example freshness queries.

as defined in §2.3.2.

Intuitively, the goal of authentication queries is to ask whether Bob will rely on some value `e1` in an important protocol operation (such as signature verification or authenticated decryption) if and only if he received that value from Alice. If Bob is successful in using `e1` for signature verification or a similar operation without it having been necessarily sent by Alice, then authentication is violated for `e1`, and the attacker was able to impersonate Alice in communicating that value.

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 re-writable 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 manipulate one message to make it seem valid in two different contexts. In passive attacker mode, a freshness query will check whether a value is "fresh" between sessions (i.e. if it has at least one composing element that is generated, non-static). In active attacker mode, it will check whether a value can be rendered "non-fresh" (i.e. static between sessions) and subsequently successfully used between sessions. In Figure 3.2, the first freshness query will be contradicted, while the second will not.

---

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

### 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. Definitions for unlinkability vary wildly despite the best efforts of researchers, but in Verifpal, we adopt the following 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."*

Based on the above, Verifpal introduced in version 0.12.0 experimental support for a notion of unlinkability based on the following checks. For an unlinkability query evaluating two values `a` and `b`:

- First, Verifpal checks to see if `a` and `b` satisfy freshness. If they do not, the query fails. Similarly to regular freshness queries, if an attacker can coerce a value to be non-fresh across sessions, then it is non-fresh and the query fails.
- If `a` and `b` both satisfy freshness, Verifpal then checks to see if the attacker can determine them as being the output of the same primitive or as having a *common source*. For example, the first and second output of the same `HKDF` construction with the same inputs. Of course, `a` and `b` can indeed be the outputs of that `HKDF` and be unlinkable; unless the attacker is able to reconstruct that same `HKDF` primitive and thereby use it to determine that both values are the outputs of it.

It is likely that these two notions are not sufficient to fully capture unlinkability between values, and future versions of Verifpal may expand this definition with additional notions, which will be similarly documented within this User Manual. 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 could be useful to check whether shared secrets derived between Alice and Bob are equivalent in all completed executions of the protocol. Equivalence queries check whether any protocol scenario can be derived such that the given values are not equivalent to one another. §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 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.

Example Unlinkability Query

```
attacker[active]
principal Alice[
  generates b
]
Alice -> Bob: b
principal Bob[
  knows private a
  generates c
  generates d
  leaks c
  h1, h2, h3 = HKDF(a, b, nil)
  h4, h5, h6 = HKDF(c, c, nil)
  h7, h8, h9 = HKDF(a, c, d)
]
queries[
  unlinkability? h1, h2, h3
  unlinkability? h4, h5, h6
  unlinkability? h7, h8, h9
]
```

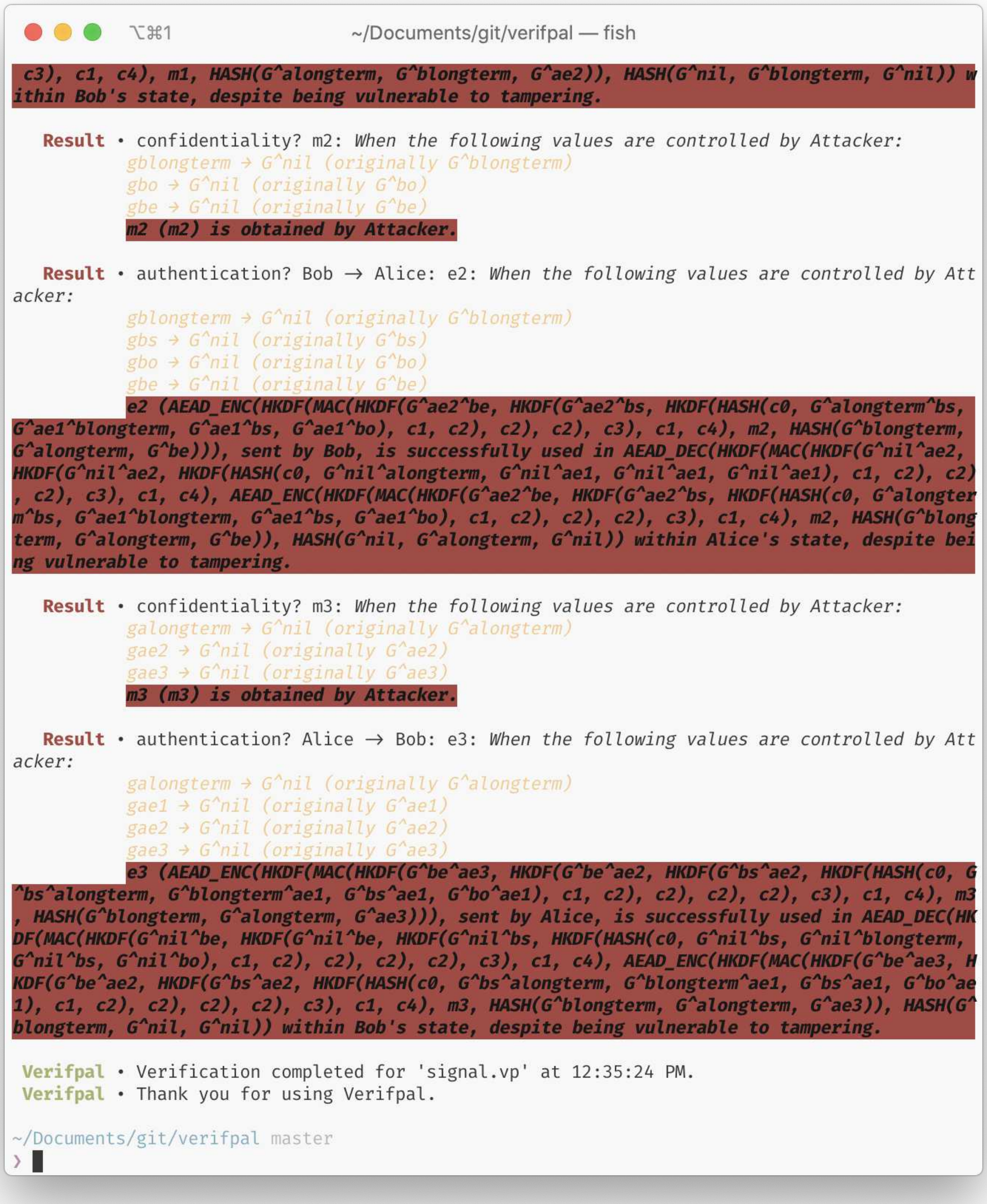
**Figure 3.3:** Example unlinkability queries.

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* [6]. 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 the same question about the future, roughly speaking.

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:



```
 c3), c1, c4), m1, HASH(G^alongterm, G^blongterm, G^ae2)), HASH(G^nil, G^blongterm, G^nil)) w
ithin Bob's state, despite being vulnerable to tampering.

   Result • confidentiality? m2: When the following values are controlled by Attacker:
            gblongterm → G^nil (originally G^blongterm)
            gbo → G^nil (originally G^bo)
            gbe → G^nil (originally G^be)
            m2 (m2) is obtained by Attacker.

   Result • authentication? Bob → Alice: e2: When the following values are controlled by Att
acker:
            gblongterm → G^nil (originally G^blongterm)
            gbs → G^nil (originally G^bs)
            gbo → G^nil (originally G^bo)
            gbe → G^nil (originally G^be)
            e2 (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^alongter
m^bs, G^ae1^blongterm, G^ae1^bs, G^ae1^bo), c1, c2), c2), c2), c3), c1, c4), m2, HASH(G^blong
term, G^alongterm, G^be)), HASH(G^nil, G^alongterm, G^nil)) within Alice's state, despite bei
ng vulnerable to tampering.

   Result • confidentiality? m3: When the following values are controlled by Attacker:
            galongterm → G^nil (originally G^alongterm)
            gae2 → G^nil (originally G^ae2)
            gae3 → G^nil (originally G^ae3)
            m3 (m3) is obtained by Attacker.

   Result • authentication? Alice → Bob: e3: When the following values are controlled by Att
acker:
            galongterm → G^nil (originally G^alongterm)
            gae1 → G^nil (originally G^ae1)
            gae2 → G^nil (originally G^ae2)
            gae3 → G^nil (originally G^ae3)
            e3 (AEAD_ENC(HKDF(MAC(HKDF(G^be^ae3, HKDF(G^be^ae2, HKDF(G^bs^ae2, HKDF(HASH(c0, G
^bs^alongterm, G^blongterm^ae1, G^bs^ae1, G^bo^ae1), c1, c2), c2), c2), c2), c3), c1, c4), m3
, HASH(G^blongterm, G^alongterm, G^ae3))), sent by Alice, is successfully used in AEAD_DEC(HK
DF(MAC(HKDF(G^nil^be, HKDF(G^nil^be, HKDF(G^nil^bs, HKDF(HASH(c0, G^nil^bs, G^nil^blongterm,
G^nil^bs, G^nil^bo), c1, c2), c2), c2), c2), c3), c1, c4), AEAD_ENC(HKDF(MAC(HKDF(G^be^ae3, H
KDF(G^be^ae2, HKDF(G^bs^ae2, HKDF(HASH(c0, G^bs^alongterm, G^blongterm^ae1, G^bs^ae1, G^bo^ae
1), c1, c2), c2), c2), c2), c3), c1, c4), m3, HASH(G^blongterm, G^alongterm, G^ae3)), HASH(G^
blongterm, G^nil, G^nil)) within Bob's state, despite being vulnerable to tampering.

 Verifpal • Verification completed for 'signal.vp' at 12:35:24 PM.
 Verifpal • Thank you for using Verifpal.

~/Documents/git/verifpal master
❯ 
```

**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.

**Query Options Example**

```
attacker[active]
principal Bob[
    knows private psk
    generates m
    e = ENC(psk, m)
    h = MAC(psk, e)
]
Bob → Alice: e, h
principal Alice[
    knows private psk
    _ = ASSERT(MAC(psk, e), h)?
    m2 = DEC(psk, e)
]
Alice → Carol: [m2]
principal Carol[
  _ = HASH(m2)
]
```

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

**Query Options Example (Cont.)**

```
queries[
    authentication? Bob → Alice: e[
    precondition[Alice → Carol: m2]
    ]
]
```

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 their viewpoint as an attacker on the network. As a passive attacker, Verifpal can only do this by deconstructing the values made available as they are shared between principals, and potentially reconstructing them into different values. As an active attacker, Verifpal can modify unguarded values as they cross the network. Each modification could result in learning new values, so an unbounded number of modifications can occur over an unbounded number of protocol executions. *“Fresh”* (i.e. generated) values are not kept across different protocol executions, as they are assumed to be different for every session of the protocol.

An active attacker can also generate their own values, such as a key pair that they control, and fabricate new values that they use as substitutes for any unguarded values sent between principals. If, during a protocol execution, a checked primitive fails, that session execution is aborted and the attacker moves on to the next one. However, values obtained thus far in that particular session execution are kept.

Verifpal also keeps track of which values are used where, the path a value takes until it arrives into the state of a principal, and who first declared or generated a value. This information is used in order to analyze for contradictions to authentication queries.

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

---

[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.

in detail, since they are salient to our analysis of Signal in Verifpal.

## 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.

Challenge-Response Protocol

```
attacker[active]
principal Server [
  knows private s
  gs = G^s
]
principal Client[
  knows private c
  gc = G^c
  generates nonce
]
Client → Server: nonce
principal Server[
  proof = SIGN(s, nonce)
]
Server → Client: gs, proof
principal Client[
  valid = SIGNVERIF(gs, nonce, proof)
  generates attestation
  signed = SIGN(c, attestation)
]
Client → Server: [gc], attestation, signed
principal Server[
  storage = SIGNVERIF(gc, attestation, signed)?
]
queries[
  authentication? Server → Client: proof
  authentication? Client → Server: signed
]
```

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

## 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 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.

---

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

[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 verifier; unlike some other automated formal verification tools [4], 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 very specific scenario enacted by principals that act in a specific way. Verifpal's goal is to then 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 ANALYSIS METHODOLOGY

Verifpal's active attacker analysis methodology (Figure 4.1) follows a simple set of procedures and algorithms. The overall process is comprised of five phases:

1. **Gather values.** Attacker passively observes a protocol execution and gathers all values shared publicly between principals.
2. **Insert learned values into attacker state.** Attacker's state ($\mathcal{V}_A$) obtains newly learned values.
3. **Apply transformations.** Attacker applies the four main *"transformations"* on all obtained values (these transformations are detailed below.)
4. **Prepare mutations for next session.** If the attacker has learned new values due to the transformations executed in the previous step, they create a combinatorial table of all possible value substitutions, and from that, derive a set of all possible value substitutions across future executions of the protocol on the network.
5. **Iterate across protocol mutations.** Attacker proceeds to execute the protocol across sessions, each time *"mutating"* the execution by mayor-in-the-middling a value. Attacker then returns to step 1 of this list. The process continues so long as the attacker keeps learning new values.

After each phase, Verifpal checks to see if it has found a contradiction to any of the queries

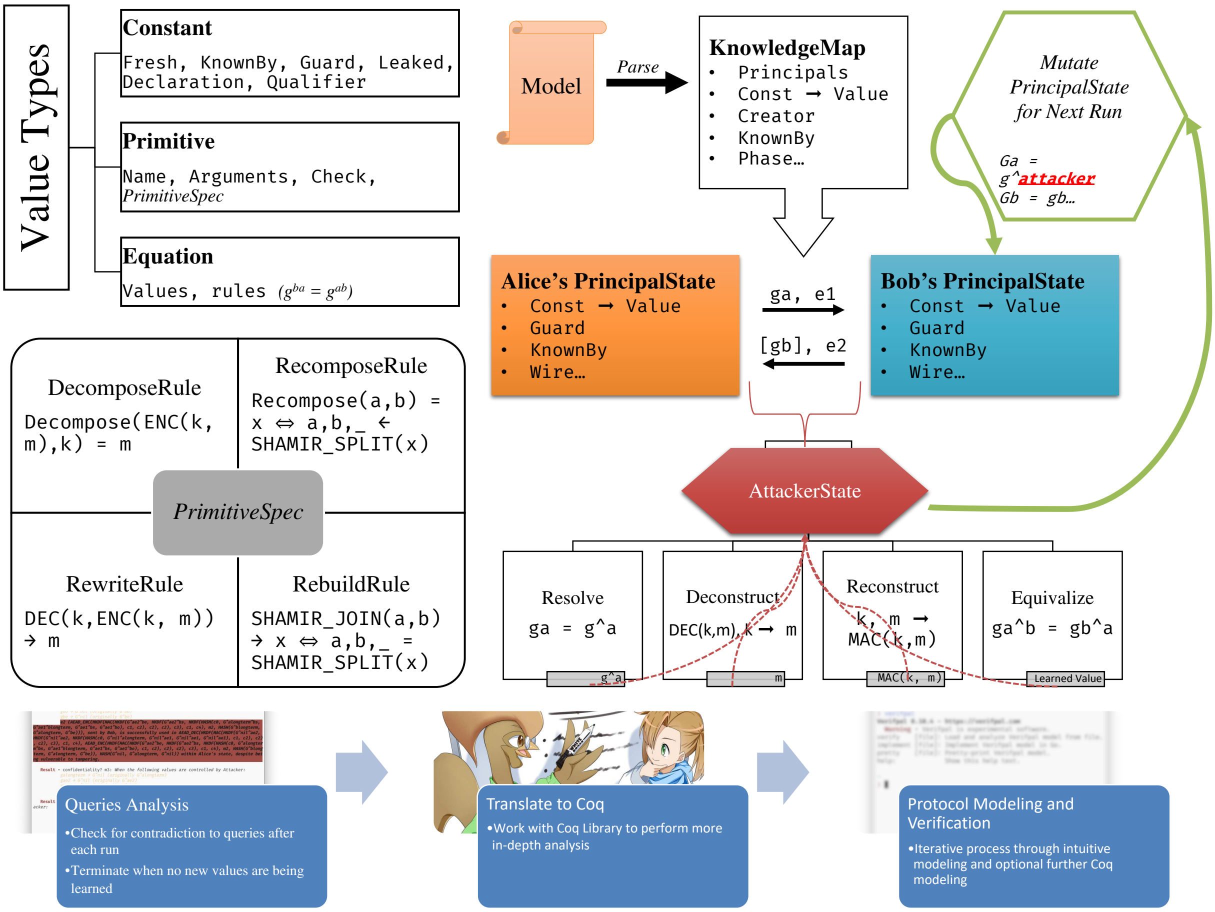

Value Types
Constant
Fresh, KnownBy, Guard, Leaked, Declaration, Qualifier
Primitive
Name, Arguments, Check, PrimitiveSpec
Equation
Values, rules ($g^{ba} = g^{ab}$)
DecomposeRule
Decompose(ENC(k, m),k) = m
RecomposeRule
Recompose(a,b) = x ⇔ a,b,_ ← SHAMIR_SPLIT(x)
PrimitiveSpec
RewriteRule
DEC(k,ENC(k, m)) → m
RebuildRule
SHAMIR_JOIN(a,b) → x ⇔ a,b,_ = SHAMIR_SPLIT(x)
Model
Parse
KnowledgeMap
Principals
Const → Value
Creator
KnownBy
Phase…
Mutate PrincipalState for Next Run
Ga = g^attacker
Gb = gb…
Alice's PrincipalState
Const → Value
Guard
KnownBy
Wire…
ga, e1
[gb], e2
Bob's PrincipalState
Const → Value
Guard
KnownBy
Wire…
AttackerState
Resolve
ga = g^a
g^a
Deconstruct
DEC(k,m), k → m
m
Reconstruct
k, m → MAC(k,m)
MAC(k, m)
Equivalize
ga^b = gb^a
Learned Value
Queries Analysis
•Check for contradiction to queries after each run
•Terminate when no new values are being learned
Translate to Coq
•Work with Coq Library to perform more in-depth analysis
Protocol Modeling and Verification
•Iterative process through intuitive modeling and optional further Coq modeling

**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 *"knowledge map"* structure 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 four transformations discussed. The attacker keeps doing this until it is unable to learn new values, at which point it mutates the model in each possible way while still following the optimization heuristics touched upon in §4.2. At the bottom, we see a description of the envisioned workflow implicit to using Verifpal in production.

specified in the model and informs the user if such a contradiction is found. The four main transformations mentioned above are the following:

- RESOLVE. Resolves a certain constant to its assigned value (for example, a primitive or an equation). Executed on $\mathcal{V}_A$, the set of all values known by the attacker.
- DECONSTRUCT. Attempts to deconstruct a primitive or an equation. In order to deconstruct a primitive, the attacker must possess sufficient values to satisfy the primitive's rewrite rule. For example, the attacker must possess `k` and `e` in order to obtain `m` by deconstructing `e = ENC(k, m)` with `k`. In order to reconstruct an equation, the attacker must similarly possess all but one private exponent. Executed on $\mathcal{V}_A$, the set of all values known by the attacker.
- RECONSTRUCT. Attempts to reconstruct primitives and equations given that the attacker possesses all of the component values. Executed on $\mathcal{V}_A$, the set of all values known by the attacker.

attacker, as well as on $\mathcal{V}_P$, the values known by the principal whose state is currently being evaluated by the attacker.

- Equivalize. Determines if the attacker can reconstruct or equivalize any values within $\mathcal{V}_P$ from $\mathcal{V}_A$. If so, then these equivalent values are added to $\mathcal{V}_A$.

## 4.2 PREVENTING STATE SPACE EXPLOSION

A common problem among symbolic model protocol verifiers is that for complex protocols, the space of the user states and value combinations that the verifier must assess becomes too large for the verifier to terminate in a reasonable time. Verifpal optimizes for this problem via certain heuristic techniques: first, Verifpal separates its analysis into a number of *stages* in which it gradually allows itself to modify more and more elements of principals' states. Only in later stages are the internal values of certain primitives (which are labeled *"explosive"* in their PrimitiveSpec) mutated. Verifpal also imposes other restrictions, such as limiting the maximum number of inputs and outputs of any primitive to five. Thus, Verifpal achieves unbounded state analysis, similarly to ProVerif, but also applies a set of heuristics that are hopefully more likely to achieve termination in a more reasonable time for large models (such as those seen for TLS 1.3 or Signal with more than three messages). Verifpal also leverages multi-threading and other such techniques to achieve faster analysis. Verifpal's stages segment its search strategy in essentially the following way, with the aim to hold back infinite mutation recursion depth as far as possible, unless queries cannot be contradicted without it:

- **Stage 1:** All of the elements of passive attacker analysis, plus constants and equation exponents may be mutated to `nil` only and not to each other (for equations, this means that `g^a` mutates to `g^nil` but not to `g^b`).
- **Stage 2:** All of the elements of Stage 1, plus non-explosive primitives are mutated but without exceeding a call depth that is pre-determined in relation to the way in which they were employed by principals in the Verifpal model. For example, `HASH(HASH(x))` will not mutate to `HASH(HASH(HASH(y)))` (since the call depth is deeper in the mutation), and `ENC(HASH(k), G^y)` will not mutate to `ENC(PW_HASH(k), k)` (since the *"skeleton"* of the original primitive does not employ `PW_HASH`, but `HASH`, and employs an equation (`G^y`) as the second argument and not a constant (`k`)).
- **Stage 3:** All of the elements of Stage 2, with the inclusion of explosive primitives.
- **Stage 4:** All of the elements of Stage 3, with the addition of constants and equation exponents being replaced with one another and not just `nil`.
- **Stage 4 and beyond:** All of the elements of Stage 3, with the addition of primitives being allowed a mutation depth of $n - 3$ where $n$ represents the current Stage, so long as the resulting mutations have the same *"skeleton"* as defined in Stage 2.

## 4.3 SOUNDNESS OF RESULTS

Verifpal has so far been used in order to model TLS, Signal, Scuttlebutt, Telegram, ProtonMail and some other protocols. So far, all of its results have been in line with previous analyses of these protocols. But anecdotal evidence is not sufficient in order to declare with full confidence that Verifpal qualifies as a proven formal verification framework.

In this section, we present an outline of Verifpal's formal analysis methodology (in addition to the formalized semantics and analysis logic of the Verifpal Coq Library), such that we can say with a high degree of confidence that:

- If an attacker is unable to obtain a value `m`, then Verifpal will answer that the query passes for the protocol described in the Verifpal model.
- If an attacker cannot find more than one way in which value `e` can be communicated between principals `A` and `B` such that `B` later employs `e` as an argument to a rewrite-capable primitive or equation, then `e` will be deemed as authenticated under `A` $\rightarrow$ `B` for the protocol described in the Verifpal model.

Formally, Verifpal is unable to claim that it never misses an attack in any model that can be expressed within its language. However, our hope is that Verifpal would not miss attacks affecting models of, or resembling, "real-world protocols". Our rationale is that given Verifpal's goals, it is preferable to avoid risking non-terminating analysis in order to account for attacks that are unlikely to occur in real-world protocol constructions. This leaves us with the problematically subjective definition of what constitutes a "real-world protocol", and implies that Verifpal will for the first few years of its existence require work on grounding and expressing more clearly the constraints of the protocols which can be expressed and for which missed attacks can truly be ruled out.

Our central argument is that the analysis logic described in this section is sufficient in order to capture a majority of confidentiality and authentication attacks within the language.

### 4.3.1 *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).

### 4.3.2 *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` and `DEC` 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 simply make it such that failed rewrites abort session execution at that point. `ASSERT`, `SPLIT`, `SIGNVERIF`, `DEC`, `AEAD_DEC`, `PKE_DEC` and other primitives have rewrite rules.

### 4.3.3 *Genealogy of Values*

In Verifpal, once a constant is known, generated or assigned, an immutable *creator* value is assigned to it defining the principal responsible for creating it. As the value travels across the network, a *sender* chain is built tracking its genealogy. For example, if Alice creates a value `m` and sends it to Bob, and if Bob then sends it to Carol, then `m` would have Alice as its creator and a sender chain of `Alice` → `Bob` → `Carol`.

When an attacker is tasked with contradicting an authentication query, it attempts to find out if a scenario exists in which a value is used in a primitive (or worse, triggers a valid rewrite rule) that does not follow the sender chain decreed by the authentication query.

### 4.3.4 *Mutations and Guarded Constants*

Except for guarded constants, the attacker can, at will, substitute any constant with any other, including constants crafted by the attacker. The goal of these substitutions is to execute the protocol in every possible permutation of constant-to-value assignments based on the values known by the attacker. Each unguarded constant risks being permuted with:

- **Other constants and values from the protocol** that have been revealed to the attacker.
- **New primitive and equation declarations** constructed from values that have been revealed to the attacker.
- **Malicious values** crafted by the attacker, including for example malicious public keys or malicious signatures under key pairs generated and owned by the attacker.

As noted earlier, once the attacker gains new values through this process, the permutation table is recalculated and the set of executions begins anew. Protocol analysis ends when no new values are known to the attacker after a complete run of all possible permutations. The goal of this step is to obtain a full search of all runs of the protocol under all possible discoverable values, given the assumption that Verifpal's analysis methodology allows the attacker to obtain all obtainable values.

Mutations and transformations are executed recursively. That is, if executing any one of RESOLVE, DECONSTRUCT, RECONSTRUCT and EQUIVALIZE leads to new values being discovered, then that transformation is executed recursively until no new values are found. If any new values are found, the series of four transformations is also re-executed recursively in its totality until no new values are obtainable by the attacker. Once that is the case, we move on to the next mutation.

* * *

*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.

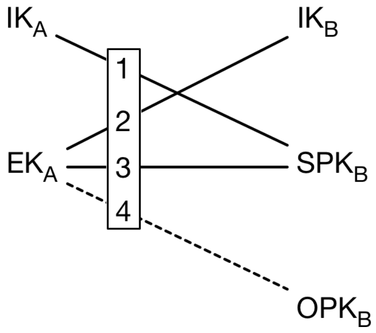


**Figure 5.1:** Signal's *"X3DH"* authenticated key exchange. $IK_A$ and $IK_B$ represent Alice and Bob's long-term key pairs. $EK_A$ represents Alice's ephemeral session key pair. $SPK_B$ and $OPK_B$ represent Bob's signed ephemeral pre-key and one-time ephemeral pre-key. Three Diffie-Hellman shared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

```
attacker[active]
principal Alice[
  knows public c0, c1, c2, c3, c4
  knows private alongterm
  galongterm = G^alongterm
]
```

Signal: Initializing Bob

```
principal Bob[
  knows public c0, c1, c2, c3, c4
  knows private blongterm, bs
  generates bo
  gblongterm = G^blongterm
  gbs = G^bs
  gbo = G^bo
  gbssig = SIGN(blongterm, gbs)
]
```

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

```
Bob → Alice: [gblongterm], gbssig, gbs, gbo
principal Alice[
  generates ae1
  gae1 = G^ae1
  amaster = HASH(c0, gbs^alongterm, gblongterm^ae1, gbs^ae1, gbo^ae1)
  arkba1, ackba1 = HKDF(amaster, c1, c2)
]
```

### 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:

---

[5] Signed pre-keys are rotated roughly once a week, while one-time pre-keys are only used once. This is simply because signing is a slow and computationally expensive process, and having Bob's phone sign every one-time pre-key (of which a server could store hundreds at a time) would be somewhat inefficient.

Signal: Alice Encrypts Message 1 to Bob

```
principal Alice[
  generates m1, ae2
  gae2 = G^ae2
  valid = SIGNVERIF(gblongterm, gbs, gbssig)?
  akshared1 = gbs^ae2
  arkab1, ackab1 = HKDF(akshared1, arkba1, c2)
  akenc1, akenc2 = HKDF(MAC(ackab1, c3), c1, c4)
  e1 = AEAD_ENC(akenc1, m1, HASH(galongterm, gblongterm, gae2))
]
Alice -> 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

```
principal Bob[
  bmaster = HASH(c0, galongterm^bs, gae1^blongterm, gae1^bs, gae1^bo)
  brkba1, bckba1 = HKDF(bmaster, c1, c2)
]
```

`Signal: Bob Decrypts Alice's Message 1`

```
principal Bob[
  bkshared1 = gae2^bs
  brkab1, bckab1 = HKDF(bkshared1, brkba1, c2)
  bkenc1, bkenc2 = HKDF(MAC(bckab1, c3), c1, c4)
  m1_d = AEAD_DEC(bkenc1, e1, HASH(galongterm, gblongterm, gae2))
]
```

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`

```
principal Bob[
  generates m2, be
  gbe = G^be
  bkshared2 = gae2^be
  brkba2, bckba2 = HKDF(bkshared2, brkab1, c2)
  bkenc3, bkenc4 = HKDF(MAC(bckba2, c3), c1, c4)
  e2 = AEAD_ENC(bkenc3, m2, HASH(gblongterm, galongterm, gbe))
]
Bob → 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`

```
principal Alice[
  akshared2 = gbe^ae2
  arkba2, ackba2 = HKDF(akshared2, arkab1, c2)
  akenc3, akenc4 = HKDF(MAC(ackba2, c3), c1, c4)
  m2_d = AEAD_DEC(akenc3, e2, HASH(gblongterm, galongterm, gbe))
]
```

`Signal: Alice Encrypts Message 3 to Bob`

```
principal Alice[
  generates m3, ae3
  gae3 = G^ae3
  akshared3 = gbe^ae3
  arkab3, ackab3 = HKDF(akshared3, arkba2, c2)
  akenc5, akenc6 = HKDF(MAC(ackab3, c3), c1, c4)
  e3 = AEAD_ENC(akenc5, m3, HASH(gblongterm, galongterm, gae3))
]
Alice → Bob: gae3, e3
```

Signal: Bob Decrypts Message 3

```
principal Bob[
  bkshared3 = gae3^be
  brkab3, bckab3 = HKDF(bkshared3, brkba2, c2)
  bkenc5, bkenc6 = HKDF(MAC(bckab3, c3), c1, c4)
  m3_d = AEAD_DEC(bkenc5, e3, HASH(gblongterm, galongterm, gae3))
]
```

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

```
  phase[1]

  principal Alice[leaks alongterm]
  principal 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

```
queries[
  confidentiality? m1
  authentication? Alice->Bob: e1
  confidentiality? m2
  authentication? Bob->Alice: e2
  confidentiality? m3
  authentication? Alice->Bob: e3
]
```

Now, let's look at our initial results:

Signal: Initial Analysis Results

```
Verifpal! 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 [15, 16]: 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

```
  Result • authentication? Bob → Alice: e2: When the following values are controlled by
      Attacker:
  gblongterm → G^nil (originally G^blongterm)
  gbs → G^nil (originally G^bs)
  gbo → G^nil (originally G^bo)
  gbe → G^nil (originally G^be)
  e2 (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.
(Analysis 25)
Result • confidentiality? m1: When the following values are controlled by Attacker:
  gblongterm → G^nil (originally G^blongterm)
  gbs → G^nil (originally G^bs)
  gbo → G^nil (originally G^bo)
  gbe → G^nil (originally G^be)
  m1 (m1) is obtained by Attacker.
(Analysis 26)
Result • confidentiality? m3: When the following values are controlled by Attacker:
  gblongterm → G^nil (originally G^blongterm)
  gbs → G^nil (originally G^bs)
  gbo → G^nil (originally G^bo)
  gbe → G^nil (originally G^be)
  m3 (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.

CHAPTER 6

# 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`

```
principal Alice[
  knows public null
  knows private n
  knows private longTermA
  generates ephemeralA
  longTermAPub = G^longTermA
  ephemeralAPub = G^ephemeralA
]
principal Bob[
  knows public null
  knows private n
  knows private longTermB
  generates ephemeralB
  longTermBPub = G^longTermB
  ephemeralBPub = G^ephemeralB
]
Bob → 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`

```
principal Alice[
  nMacAlice = MAC(n, ephemeralAPub)
]
Alice → Bob: ephemeralAPub, nMacAlice
principal Bob[
  nMacAliceValid = ASSERT(MAC(n, ephemeralAPub), nMacAlice)?
  nMacBob = MAC(n, ephemeralBPub)
]
Bob → 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

```
principal Alice[
  nMacBobValid = ASSERT(MAC(n, ephemeralBPub), nMacBob)?
  ephemeralSecretAlice = ephemeralBPub^ephemeralA
  longTermSecretAlice = longTermBPub^ephemeralA
  masterSecret1Alice = HASH(n, ephemeralSecretAlice, longTermSecretAlice)
  sig1Alice = SIGN(longTermA, HASH(n, longTermBPub, ephemeralSecretAlice))
  secretBox1Alice = AEAD_ENC(masterSecret1Alice, sig1Alice, null)
  secretBox2Alice = AEAD_ENC(masterSecret1Alice, longTermAPub, null)
  longEphemeralSecretAlice = ephemeralBPub^longTermA
  masterSecret2Alice = HASH(n, ephemeralSecretAlice, longTermSecretAlice,
      longEphemeralSecretAlice)
]
Alice → Bob: secretBox1Alice, secretBox2Alice
```

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

Scuttlebutt: Bob Generates Session Secrets

```
principal Bob[
  ephemeralSecretBob = ephemeralAPub^ephemeralB
  longTermSecretBob = ephemeralAPub^longTermB
  masterSecret1Bob = HASH(n, ephemeralSecretBob, longTermSecretBob)
  sig1Bob = AEAD_DEC(masterSecret1Bob, secretBox1Alice, null)?
  longTermAPub_Bob = AEAD_DEC(masterSecret1Bob, secretBox2Alice, null)?
  sig1Valid = SIGNVERIF(longTermAPub_bob, HASH(n, longTermBPub, ephemeralSecretBob),
      sig1Bob)?
  longEphemeralSecretBob = longTermAPub_Bob^ephemeralB
]
```

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

Scuttlebutt: Bob Signs Session Transcript

```
principal Bob[
  sig2Bob = SIGN(longTermB, HASH(n, sig1Bob, longTermAPub_Bob, ephemeralSecretBob))
  masterSecret2Bob = HASH(n, ephemeralSecretBob, longTermSecretBob,
      longEphemeralSecretBob)
  secretBox1Bob = AEAD_ENC(masterSecret2Bob, sig2Bob, null)
]
Bob → 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

```
principal Alice[
  knows private m1
  sig2Alice = AEAD_DEC(masterSecret2Alice, secretBox1Bob, null)?
  sig2Valid = SIGNVERIF(longTermBPub, HASH(n, sig1Alice, longTermAPub,
      ephemeralSecretAlice), sig2Alice)?
  secretBoxM1Alice = AEAD_ENC(masterSecret2Alice, m1, null)
]
Alice → Bob: secretBoxM1Alice
```

Scuttlebutt: Bob Receives and Decrypts Message from Alice

```
principal Bob[
  knows private m2
  m1Bob = AEAD_DEC(masterSecret2Bob, secretBoxM1Alice, null)?
  secretBoxM2Bob = AEAD_ENC(masterSecret2Bob, m2, null)
]
```

Scuttlebutt: Bob Encrypts and Sends Message to Alice

```
Bob → Alice: secretBoxM2Bob
principal Alice [
  m2Alice = AEAD_DEC(masterSecret2Alice, secretBoxM2Bob, null)?
]
```

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

queries[
  confidentiality? m1
  confidentiality? m2
  confidentiality? longTermAPub
  authentication? Alice → Bob: secretBox1Alice
  authentication? Alice → Bob: secretBox2Alice
  authentication? Bob → Alice: secretBox1Bob
  authentication? Alice → Bob: secretBoxM1Alice
  authentication? Bob → Alice: secretBoxM2Bob
]
```

Now, let's look at our initial results:

```
Scuttlebutt: Initial Results

Verifpal! 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

Verifpal! 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:

```
Scuttlebutt: Results with Public n and Bob MitM

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

---

[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: Alice Leaks Long-Term Private Key

Alice → 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

Result! confidentiality? n: n is obtained by the attacker as n
Result! 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 [17] (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

```
attacker[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

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

  knows public BroadcastKey

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

  generates SK0A

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

  EphID00A, EphID01A, EphID02A = HKDF(nil, SK0A, BroadcastKey)
]
```

The same thing goes for Bob, and Charlie:

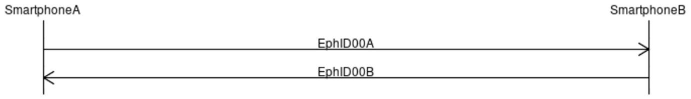
**DP-3T: SmartphoneB, SmartphoneC Setup**

```
principal SmartphoneB[
  knows public BroadcastKey
  generates SK0B
  EphID00B, EphID01B, EphID02B = HKDF(nil, SK0B, BroadcastKey)
]

principal SmartphoneC[
  knows public BroadcastKey
  generates SK0C
  EphID00C, EphID01C, EphID02C = HKDF(nil, SK0C, BroadcastKey)
]
```

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:



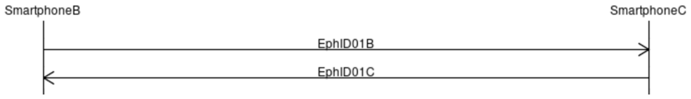
Here is how the above message exchange is modeled in Verifpal:

**DP-3T: EphID Communication**

```
// Sender -> Recipient : Name of Value

SmartphoneA -> SmartphoneB: EphID00A
SmartphoneB -> SmartphoneA: EphID00B
```

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



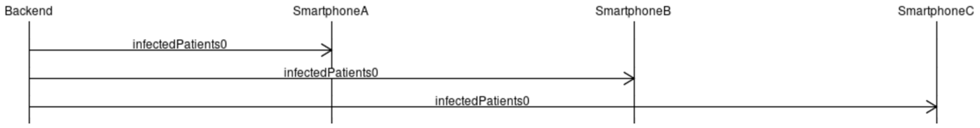
Modeling this is equally simple:

**DP-3T: EphID Communication**

```
SmartphoneC -> SmartphoneB: EphID01C
SmartphoneB -> 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**

```
principal BackendServer[
  // Let's assume that infectedPatients0 is the list of infected patients on day 0
  knows private infectedPatients0
]

BackendServer -> SmartphoneA: infectedPatients0
BackendServer -> SmartphoneB: infectedPatients0
BackendServer -> 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**

```
principal SmartphoneA[
  SK1A = HASH(SK0A)
  EphID10A, EphID11A, EphID12A = HKDF(nil, SK1A, BroadcastKey)
]

principal SmartphoneB[
  SK1B = HASH(SK0B)
  EphID10B, EphID11B, EphID12B = HKDF(nil, SK1B, BroadcastKey)
]

principal SmartphoneC[
  SK1C = HASH(SK0C)
  EphID10C, EphID11C, EphID12C = HKDF(nil, SK1C, BroadcastKey)
]
```

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

```
principal SmartphoneA[
  SK2A = HASH(SK1A)
  EphID20A, EphID21A, EphID22A = HKDF(nil, SK2A, BroadcastKey)
]
```

### 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

```
phase[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

```
principal HealthCareAuthority[
  generates triggerToken
  knows private ephemeral_sk
  m1 = ENC(ephemeral_sk, triggerToken)
]

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

HealthCareAuthority -> BackendServer : [m1]
HealthCareAuthority -> 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

```
principal SmartphoneA[
  knows private ephemeral_sk
  m1_dec = DEC(ephemeral_sk, m1)
  m2 = AEAD_ENC(ephemeral_sk, SK1A, m1_dec)
]

SmartphoneA→ 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.

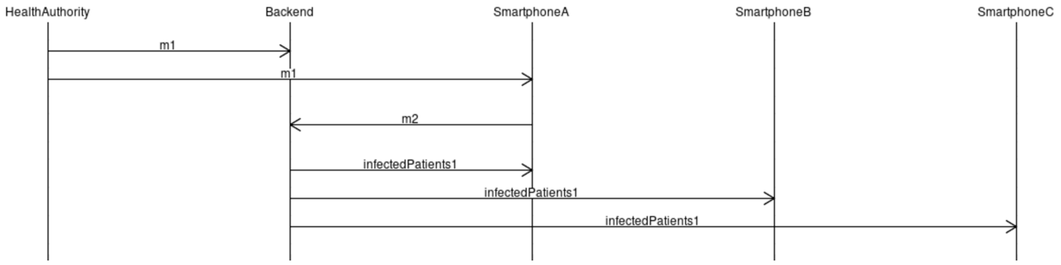
Finally, the Backend Server will add `SK1A` to the list of infected patients previously defined, and then send this list to all of the individuals in this community.

DP-3T: Updating List of Infected Patents

```
principal BackendServer [
  knows private ephemeral_sk
  m2_dec = AEAD_DEC(ephemeral_sk, m2, DEC(ephemeral_sk, m1))?
  infectedPatients1 = CONCAT(infectedPatients0, m2_dec)
]

BackendServer→ SmartphoneA: infectedPatients1
BackendServer→ SmartphoneB: infectedPatients1
BackendServer→ 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

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

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

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

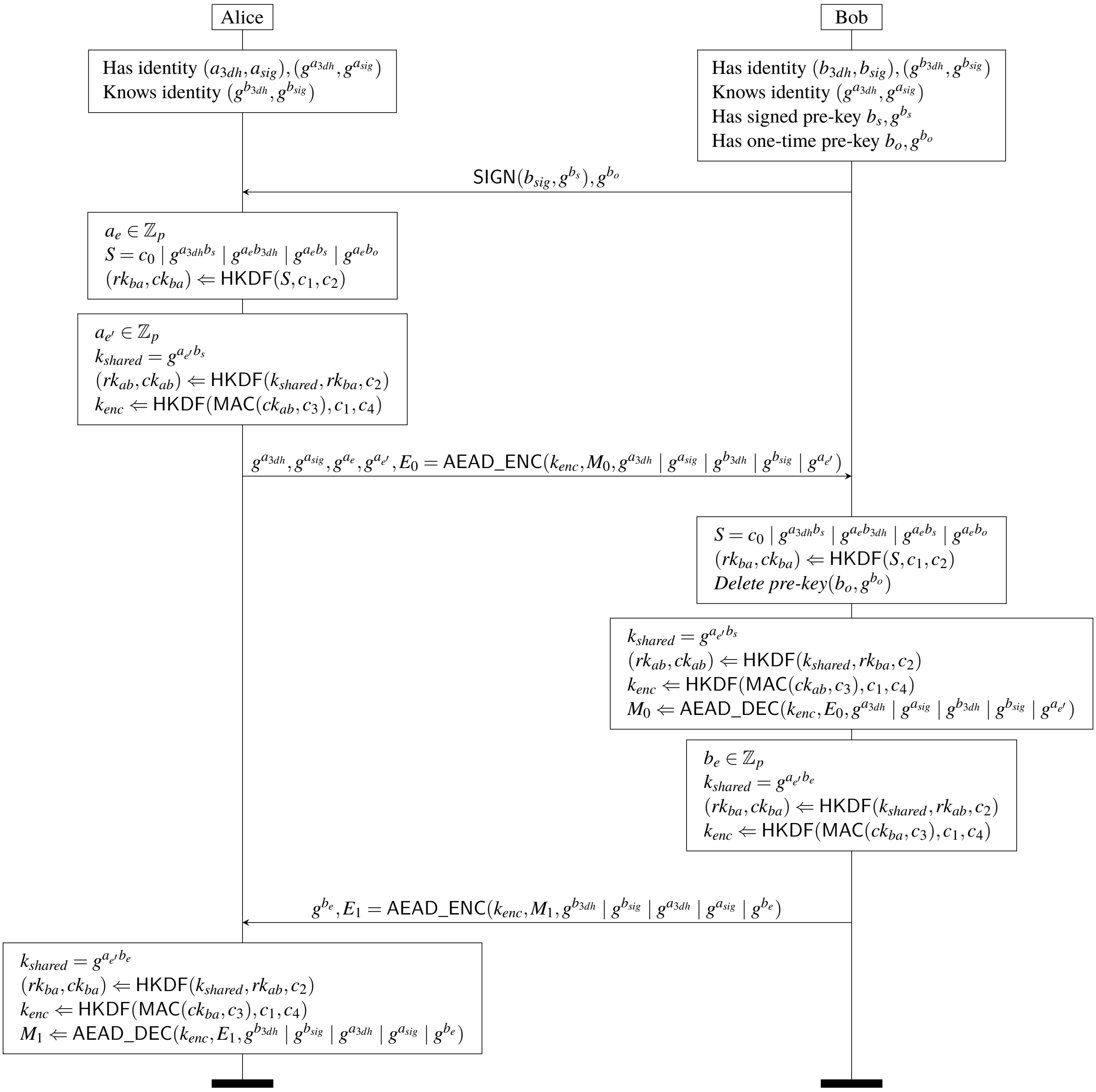

Alice
Bob
Has identity $(a_{3dh}, a_{sig}), (g^{a_{3dh}}, g^{a_{sig}})$
Knows identity $(g^{b_{3dh}}, g^{b_{sig}})$
Has identity $(b_{3dh}, b_{sig}), (g^{b_{3dh}}, g^{b_{sig}})$
Knows identity $(g^{a_{3dh}}, g^{a_{sig}})$
Has signed pre-key $b_s, g^{b_s}$
Has one-time pre-key $b_o, g^{b_o}$
$\mathsf{SIGN}(b_{sig}, g^{b_s}), g^{b_o}$
$a_e \in \mathbb{Z}_p$
$S = c_0 \mid g^{a_{3dh} b_s} \mid g^{a_e b_{3dh}} \mid g^{a_e b_s} \mid g^{a_e b_o}$
$(rk_{ba}, ck_{ba}) \Leftarrow \mathsf{HKDF}(S, c_1, c_2)$
$a_{e'} \in \mathbb{Z}_p$
$k_{shared} = g^{a_{e'} b_s}$
$(rk_{ab}, ck_{ab}) \Leftarrow \mathsf{HKDF}(k_{shared}, rk_{ba}, c_2)$
$k_{enc} \Leftarrow \mathsf{HKDF}(\mathsf{MAC}(ck_{ab}, c_3), c_1, c_4)$
$g^{a_{3dh}}, g^{a_{sig}}, g^{a_e}, g^{a_{e'}}, E_0 = \mathsf{AEAD\_ENC}(k_{enc}, M_0, g^{a_{3dh}} \mid g^{a_{sig}} \mid g^{b_{3dh}} \mid g^{b_{sig}} \mid g^{a_{e'}})$
$S = c_0 \mid g^{a_{3dh} b_s} \mid g^{a_e b_{3dh}} \mid g^{a_e b_s} \mid g^{a_e b_o}$
$(rk_{ba}, ck_{ba}) \Leftarrow \mathsf{HKDF}(S, c_1, c_2)$
$Delete\ pre\text{-}key(b_o, g^{b_o})$
$k_{shared} = g^{a_{e'} b_s}$
$(rk_{ab}, ck_{ab}) \Leftarrow \mathsf{HKDF}(k_{shared}, rk_{ba}, c_2)$
$k_{enc} \Leftarrow \mathsf{HKDF}(\mathsf{MAC}(ck_{ab}, c_3), c_1, c_4)$
$M_0 \Leftarrow \mathsf{AEAD\_DEC}(k_{enc}, E_0, g^{a_{3dh}} \mid g^{a_{sig}} \mid g^{b_{3dh}} \mid g^{b_{sig}} \mid g^{a_{e'}})$
$b_e \in \mathbb{Z}_p$
$k_{shared} = g^{a_{e'} b_e}$
$(rk_{ba}, ck_{ba}) \Leftarrow \mathsf{HKDF}(k_{shared}, rk_{ab}, c_2)$
$k_{enc} \Leftarrow \mathsf{HKDF}(\mathsf{MAC}(ck_{ba}, c_3), c_1, c_4)$
$g^{b_e}, E_1 = \mathsf{AEAD\_ENC}(k_{enc}, M_1, g^{b_{3dh}} \mid g^{b_{sig}} \mid g^{a_{3dh}} \mid g^{a_{sig}} \mid g^{b_e})$
$k_{shared} = g^{a_{e'} b_e}$
$(rk_{ba}, ck_{ba}) \Leftarrow \mathsf{HKDF}(k_{shared}, rk_{ab}, c_2)$
$k_{enc} \Leftarrow \mathsf{HKDF}(\mathsf{MAC}(ck_{ba}, c_3), c_1, c_4)$
$M_1 \Leftarrow \mathsf{AEAD\_DEC}(k_{enc}, E_1, g^{b_{3dh}} \mid g^{b_{sig}} \mid g^{a_{3dh}} \mid g^{a_{sig}} \mid g^{b_e})$

**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 $M_0$.

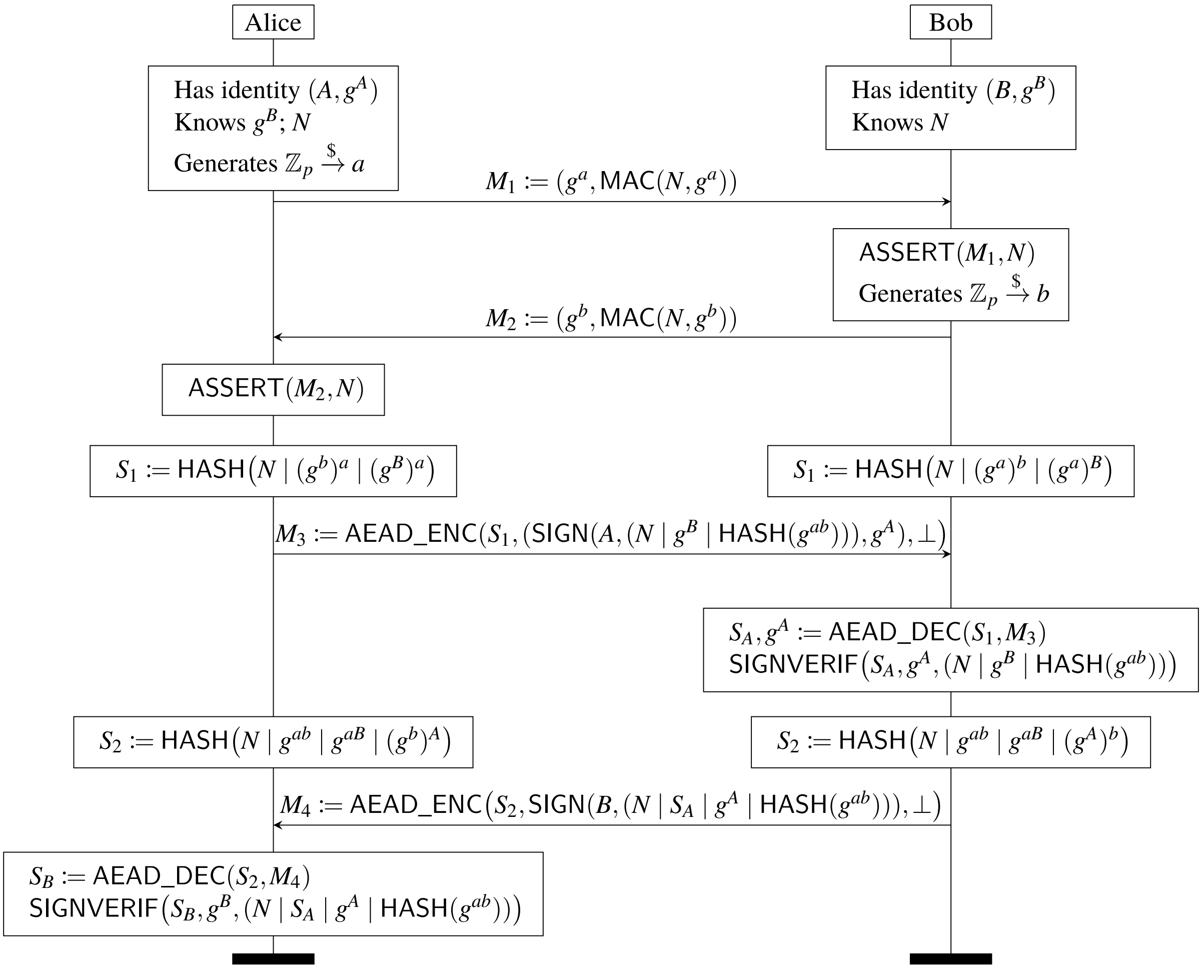

Alice
Bob
Has identity $(A, g^A)$
Knows $g^B$; $N$
Generates $\mathbb{Z}_p \xrightarrow{\$} a$
Has identity $(B, g^B)$
Knows $N$
$M_1 := (g^a, \mathsf{MAC}(N, g^a))$
$\mathsf{ASSERT}(M_1, N)$
Generates $\mathbb{Z}_p \xrightarrow{\$} b$
$M_2 := (g^b, \mathsf{MAC}(N, g^b))$
$\mathsf{ASSERT}(M_2, N)$
$S_1 := \mathsf{HASH}\big(N \mid (g^b)^a \mid (g^B)^a\big)$
$S_1 := \mathsf{HASH}\big(N \mid (g^a)^b \mid (g^a)^B\big)$
$M_3 := \mathsf{AEAD\_ENC}(S_1, (\mathsf{SIGN}(A, (N \mid g^B \mid \mathsf{HASH}(g^{ab}))), g^A), \perp)$
$S_A, g^A := \mathsf{AEAD\_DEC}(S_1, M_3)$
$\mathsf{SIGNVERIF}\big(S_A, g^A, (N \mid g^B \mid \mathsf{HASH}(g^{ab}))\big)$
$S_2 := \mathsf{HASH}\big(N \mid g^{ab} \mid g^{aB} \mid (g^b)^A\big)$
$S_2 := \mathsf{HASH}\big(N \mid g^{ab} \mid g^{aB} \mid (g^A)^b\big)$
$M_4 := \mathsf{AEAD\_ENC}\big(S_2, \mathsf{SIGN}(B, (N \mid S_A \mid g^A \mid \mathsf{HASH}(g^{ab}))), \perp\big)$
$S_B := \mathsf{AEAD\_DEC}(S_2, M_4)$
$\mathsf{SIGNVERIF}\big(S_B, g^B, (N \mid S_A \mid g^A \mid \mathsf{HASH}(g^{ab}))\big)$

**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 15 of the First Edition of the Verifpal User Manual.

*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.