Verifpal

User Manual

VERSION 2026.09.06

SYMBOLIC SOFTWARE

Nadim Kobeissi

PUBLISHED BY SYMBOLIC SOFTWARE

TYPESET ON SEPTEMBER 6, 2026

Acknowledgments

Verifpal draws on Professor Bruno Blanchet's foundational work in formal verification.[1] Toby Fox's *Undertale* also influenced the project's visual style and tone.

The NLnet Foundation supported Verifpal through the *NGI0 Privacy Enhancing Technologies Fund*. This fund was 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, grant agreement №825310.

I thank my students Georgio Nicolas, who contributed the first edition's chapter on DP-3T, and Sasha Lapiha. I also thank Vlad Antipin. All three provided feedback on earlier editions of this manual.

I also thank the artists at Collateral Damage Studios, who created the Verifpal manga and illustrations over three months of close collaboration.

[1] This acknowledgment does not imply Professor Blanchet's endorsement of Verifpal.

Wall of Honor

Verifpal is authored by Nadim Kobeissi. The people named here gave the project suggestions, bug reports, ideas and discussion. Their names are recorded with gratitude.

- Angèle Bossuat
- Bruno Blanchet *Prof. Dr.*
- Fabian Drinck
- Friedrich Wiemer
- Georgio Nicolas
- Heba Ajjour
- Jean-Philippe Aumasson *Dr.*
- Laurent Grémy
- Loup Vaillant David
- Mario Raso
- Michiel Leenars
- Mukesh Tiwari *Dr.*
- Oleksandra “Sasha” Lapiha
- Oskar Goldhahn
- Renaud Lifchitz
- Sebastian R. Verschoor
- Tom Roeder

Dedication

To Karthikeyan Bhargavan, Bruno Blanchet, Antoine Delignat-Lavaud, Graham Steel and Harry Halpin, with gratitude for their guidance and support.

To my students.

8aad60363031ce951081da9eba8f4f24

Introduction

Verifpal analyzes the security of cryptographic protocol designs. These protocols support many familiar systems. HTTPS relies on TLS to protect web connections, while WhatsApp uses the Signal protocol to protect messages.

A protocol is expected to satisfy specific *security properties*. For example, TLS should prevent an intermediary from reading a password sent to a website. It should also let the user confirm the website's identity rather than communicate with an impersonator. These properties are called *confidentiality* and *authentication*. Formal analysis of draft TLS 1.3 found flaws in these areas before the standard was deployed [1].

Manual reasoning becomes difficult as a protocol grows. Each message introduces new cases, and an attack may depend on an interaction that the designer did not consider. *Automated formal verification* addresses this problem. Tools such as ProVerif [2] and Tamarin [3] let researchers describe a protocol and test its claimed security properties against a formal adversary.[2] Figure 2 illustrates the complexity of even a simplified Signal exchange.

Verifpal is intended for readers who have limited experience with formal methods. Its models describe messages exchanged by explicit principals such as Alice and Bob, and its attack traces explain the causal steps of each result.

Protocol analysis often extends beyond confidentiality and authentication. *Forward secrecy* asks whether compromising a device exposes messages sent before the compromise. *Post-compromise security* asks whether a protocol can restore its guarantees for later messages after a compromise [7].

Other tools support properties and modeling techniques beyond Verifpal's current scope. ProVerif, for example, uses the applied pi calculus [8] and translates models into Horn clauses [9]. Verifpal makes a different tradeoff: it provides a small modeling language, built-in cryptographic primitives and attack traces intended for direct inspection. This narrower interface reduces the background needed to begin analyzing a protocol.

Verifpal follows four design principles:

- *A readable protocol language.* Models use explicit principals with independent state. Principals know and generate values, apply cryptographic primitives and exchange

[2] ProVerif, Tamarin and Verifpal analyze protocols in the *symbolic model*, also called the Dolev–Yao model [4]. This model treats cryptographic operations as ideal functions. A ciphertext can be decrypted only with the correct key, for example, and a hash cannot be inverted. The abstraction omits implementation details such as key lengths and block-cipher modes so that the analysis can focus on a protocol's logical structure. Computational verifiers such as CryptoVerif [5] instead reason about concrete cryptographic assumptions. See [6] for a comparison of the two approaches.

messages over a network. This structure resembles an informal protocol description while remaining precise enough for analysis.

- *Safe modeling defaults.* Verifpal provides a fixed set of cryptographic primitives, including `ENC`, `DEC`, `AEAD_ENC`, `AEAD_DEC` and `SIGN`. Users cannot accidentally give a standard operation an incorrect definition.[3] Constants occupy a global namespace and cannot be redefined.
- *Traceable results.* During analysis, Verifpal reports the values that the attacker can decompose, obtain and reconstruct. If an attack contradicts a query, Verifpal presents a numbered trace that relates each attacker action to the protocol execution.
- *Support for common workflows.* The Verifpal binary includes a language server used by extensions for Visual Studio Code, Neovim and Zed. These integrations provide syntax highlighting, diagnostics, documentation, formatting, diagrams and analysis. The browser-based Verifpal Workbench runs through WebAssembly and requires no local installation.

Verifpal is a protocol *analysis* tool, not a proof-producing verifier. If Verifpal reports no attack, it means that a bounded search did not find one; it does not prove that none exists. A reported attack has been re-executed and checked by a validator that is independent of the search, although two known limitations of the session model can still produce a false report. Chapter 5 defines this boundary precisely. The analysis engine is specified in a companion paper [10], which is the authority on its semantics when this manual and the paper differ.

This manual has three parts. Part I covers installation (Chapter 1), the modeling language (Chapter 2), and security queries and attacker models (Chapter 3). Part II documents the built-in primitives and their equations (Chapter 4) and the analysis engine (Chapter 5). Part III applies these concepts to Signal (Chapter 6), Scuttlebutt (Chapter 7) and a post-quantum handshake (Chapter 8).

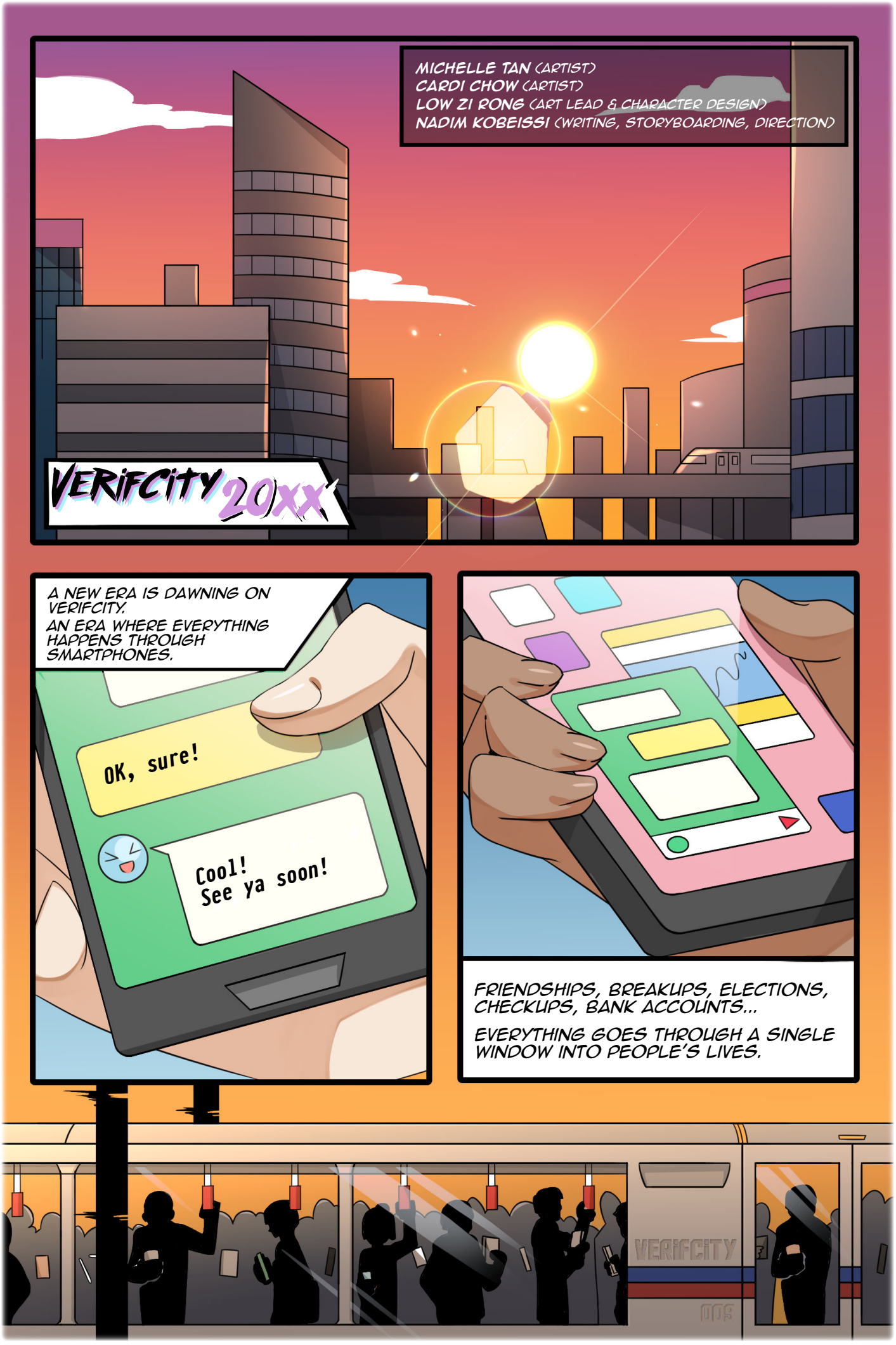
The manual assumes familiarity with hashes, encryption, Diffie–Hellman key exchange and digital signatures. No background in formal methods is required.

Readers who need an introduction to cryptography may consult Jean-Philippe Aumasson's *Serious Cryptography* or David Wong's *Real-World Cryptography* alongside this manual.

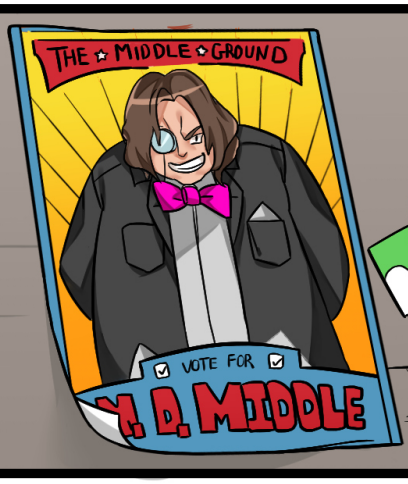
Nadim Kobeissi

September 2026

[3] User-defined primitives allow ProVerif to express models that Verifpal cannot. This is a deliberate difference in scope.

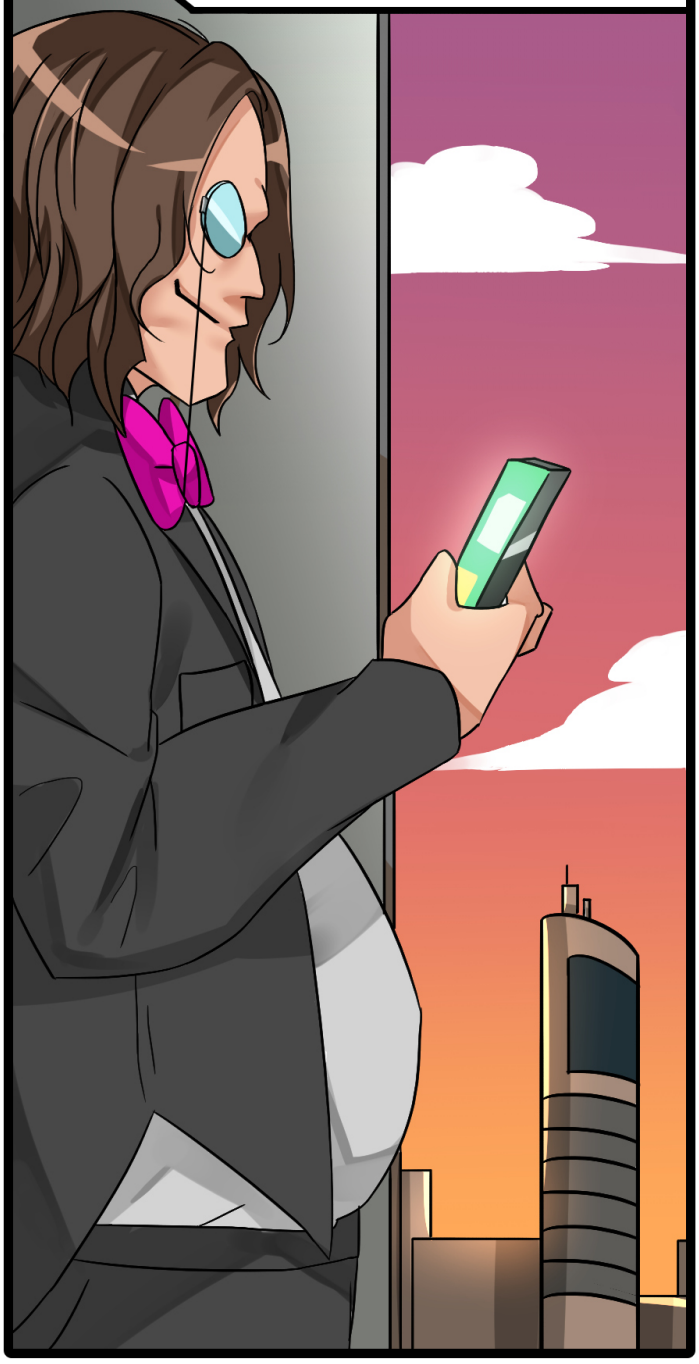

MICHELLE TAN (ARTIST)
CARDI CHOW (ARTIST)
LOW ZI RONG (ART LEAD & CHARACTER DESIGN)
NADIM KOBEISSI (WRITING, STORYBOARDING, DIRECTION)
VERIFCITY 20XX
A NEW ERA IS DAWNING ON VERIFCITY.
AN ERA WHERE EVERYTHING HAPPENS THROUGH SMARTPHONES.
OK, sure!
Cool! See ya soon!
FRIENDSHIPS, BREAKUPS, ELECTIONS,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



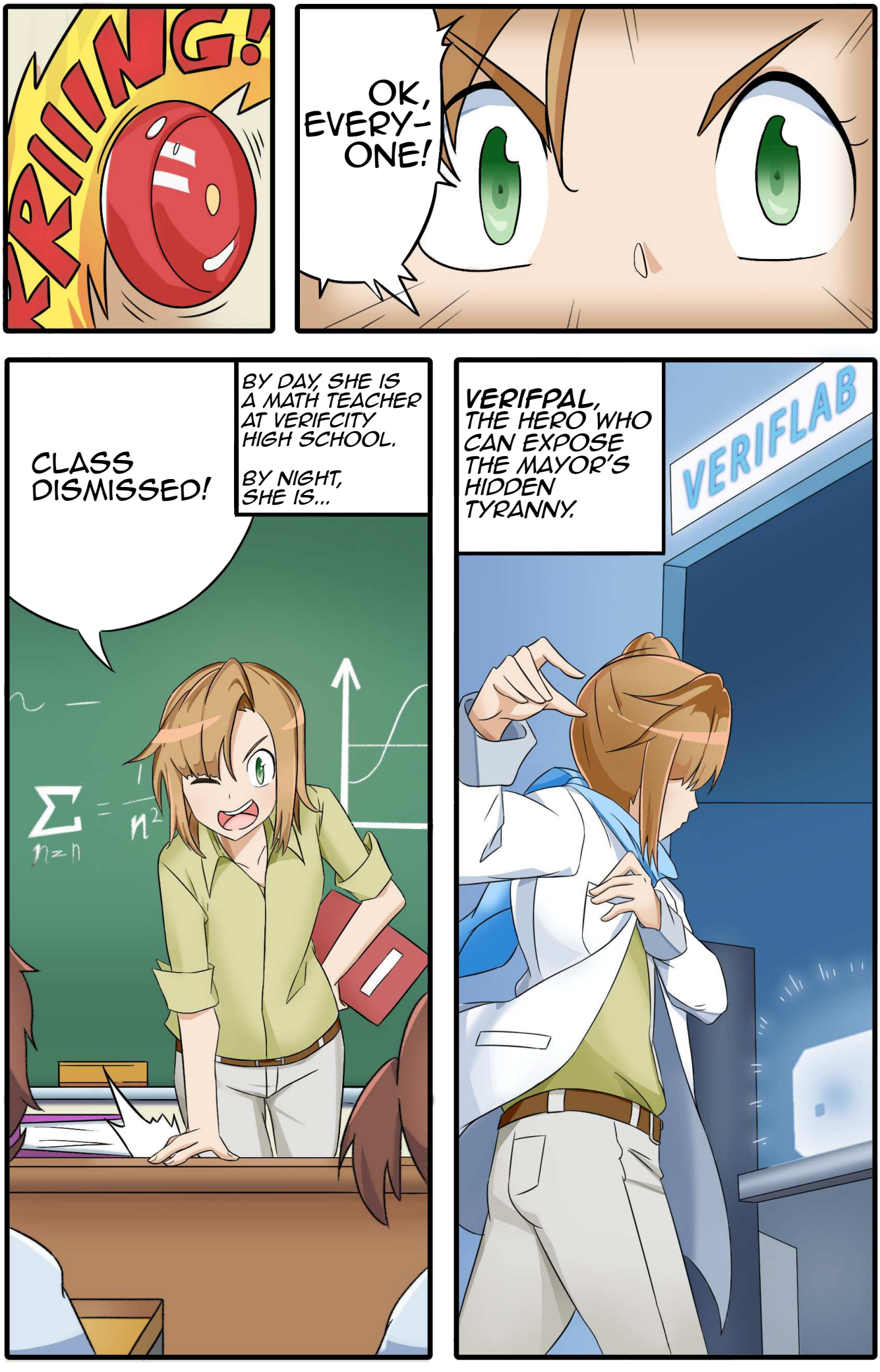
BUT INSTEAD, HE POISONED VERIFCITY'S COMMUNICATION PROTOCOLS.

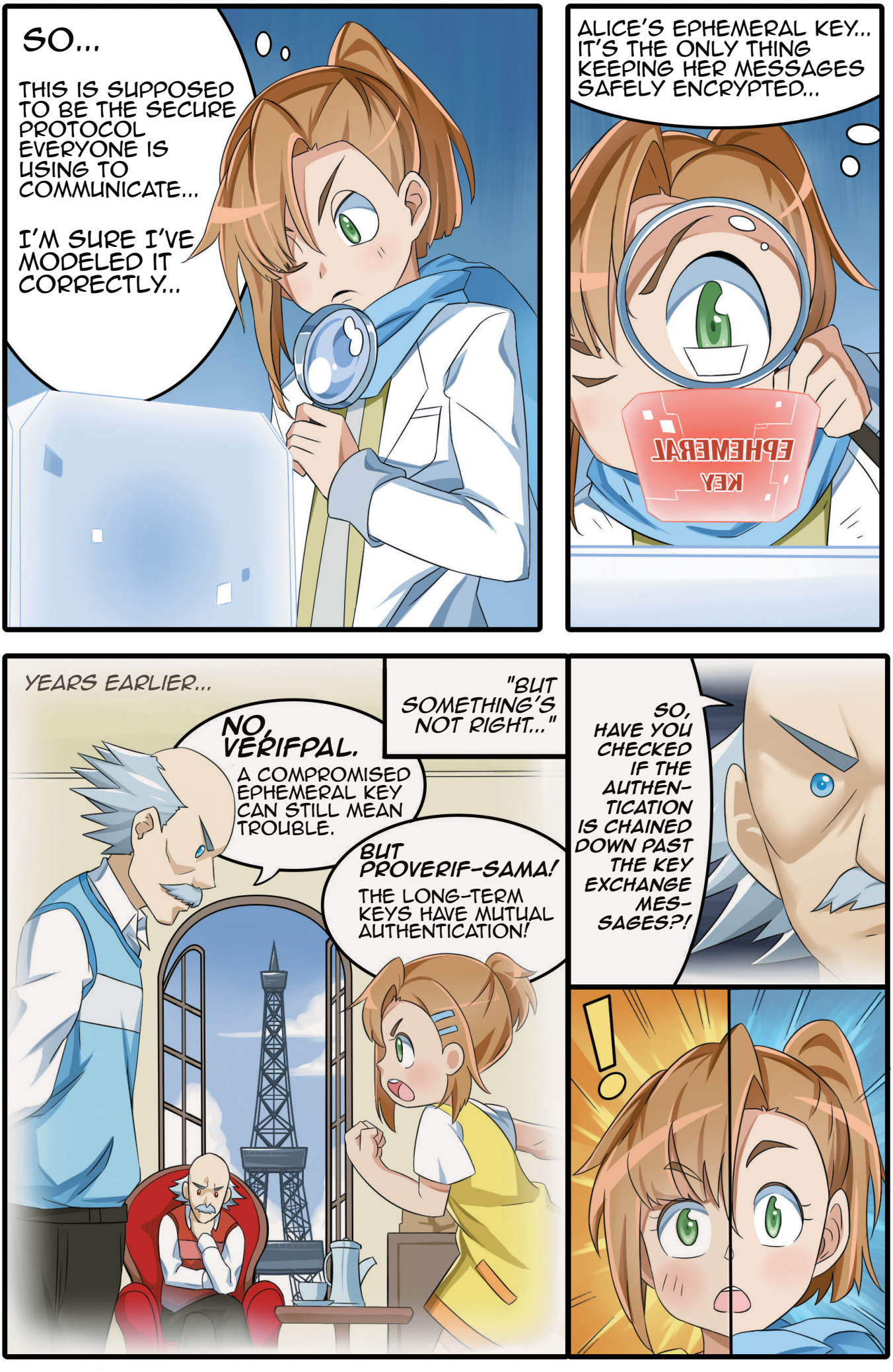
LETTING HIM MONITOR EVERYONE.

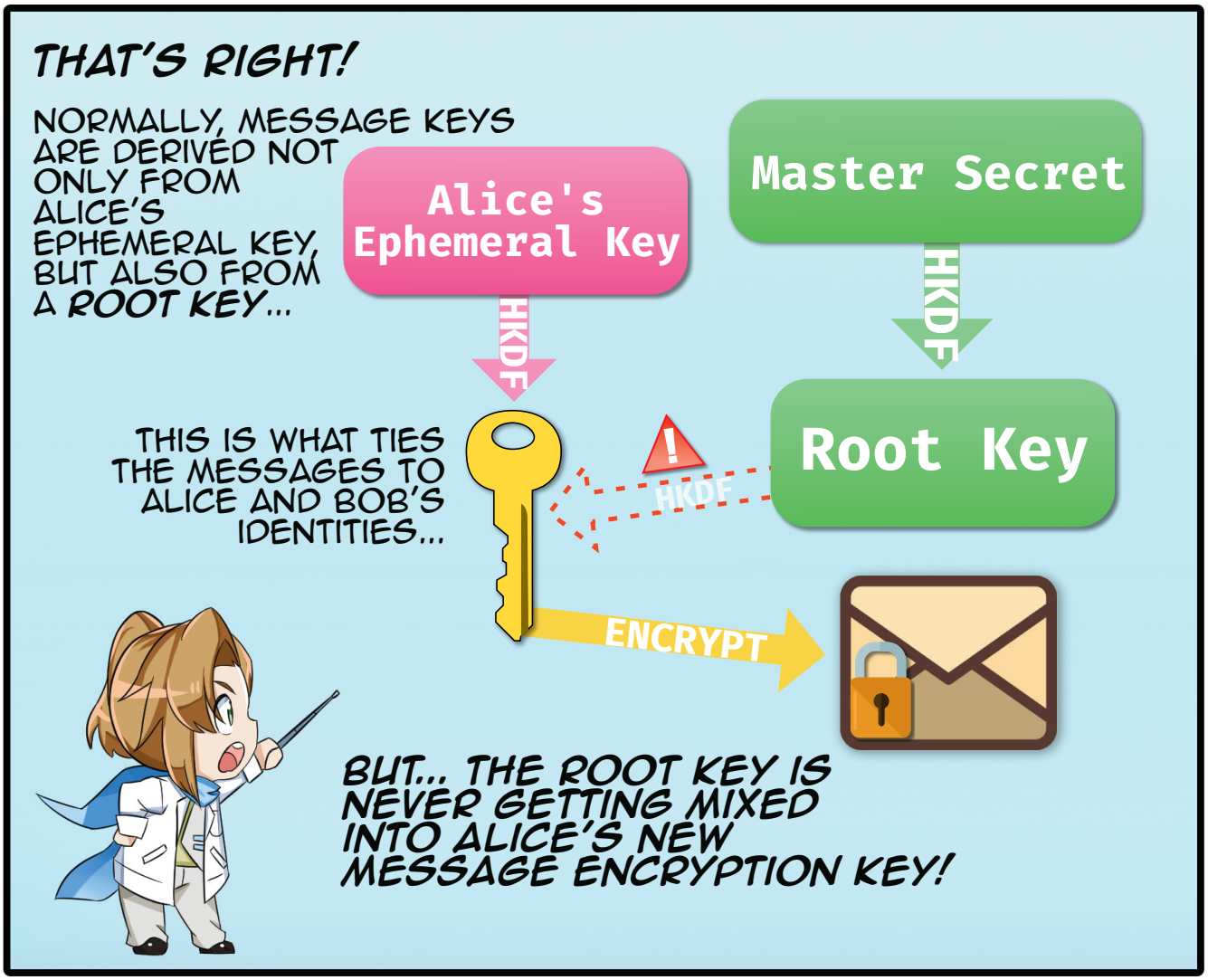


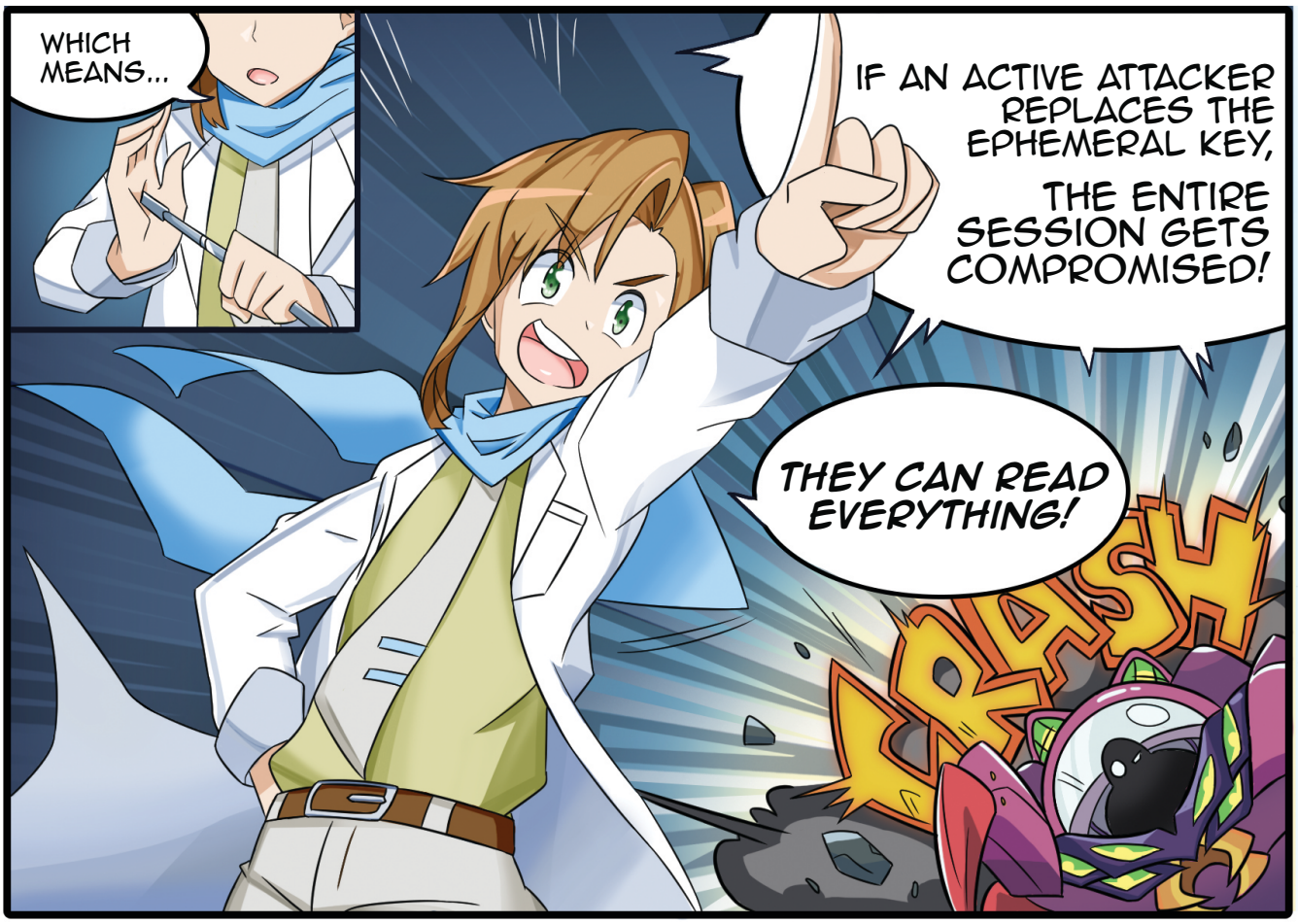
EVER SINCE HIS ELECTION, PEOPLE'S LIVES HAVE BEEN AT RISK OF EXPOSURE.
NOBODY KNOWS WHERE IT'S COMING FROM, OR WHEN IT WILL STOP.

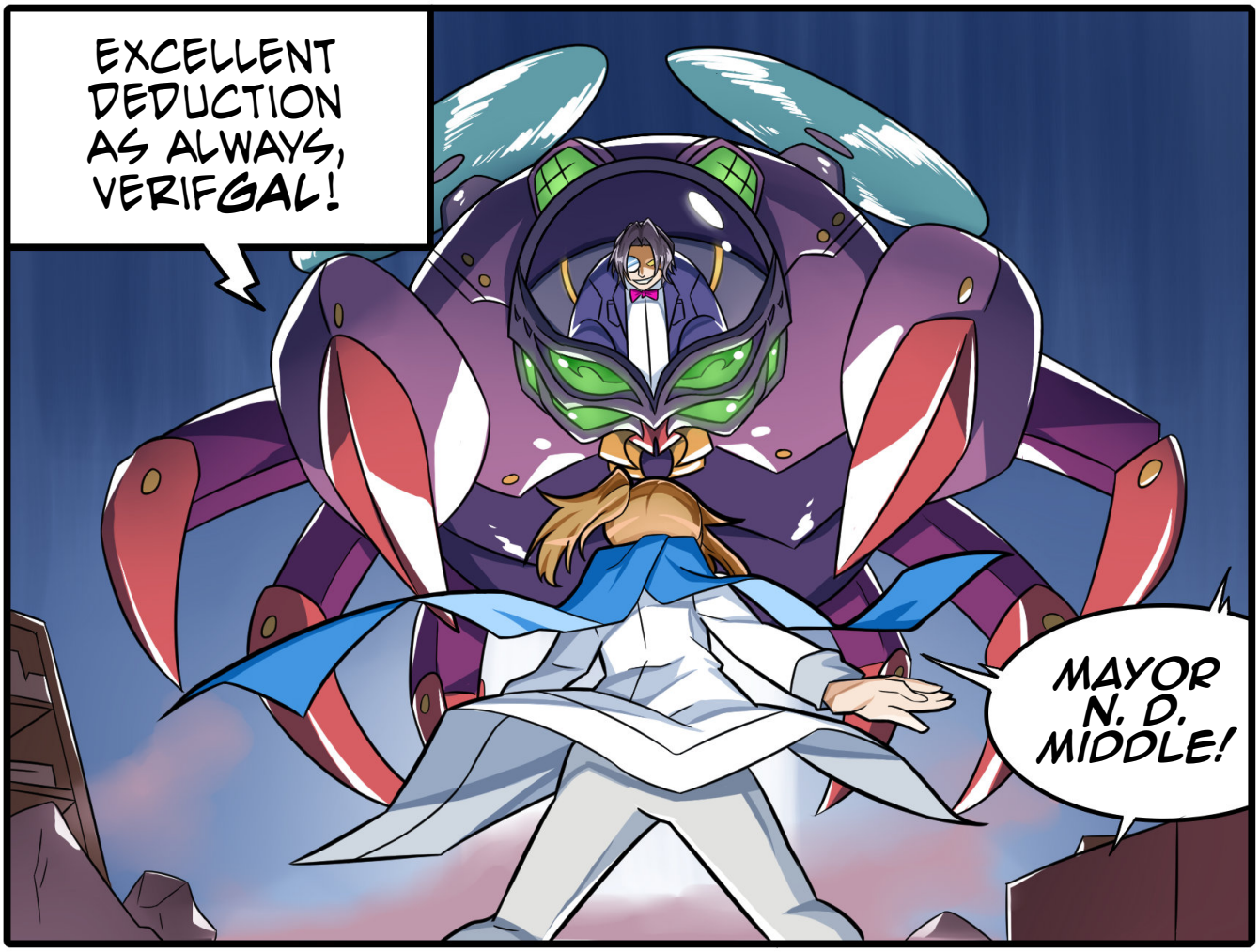
WELL, ALMOST NOBODY.

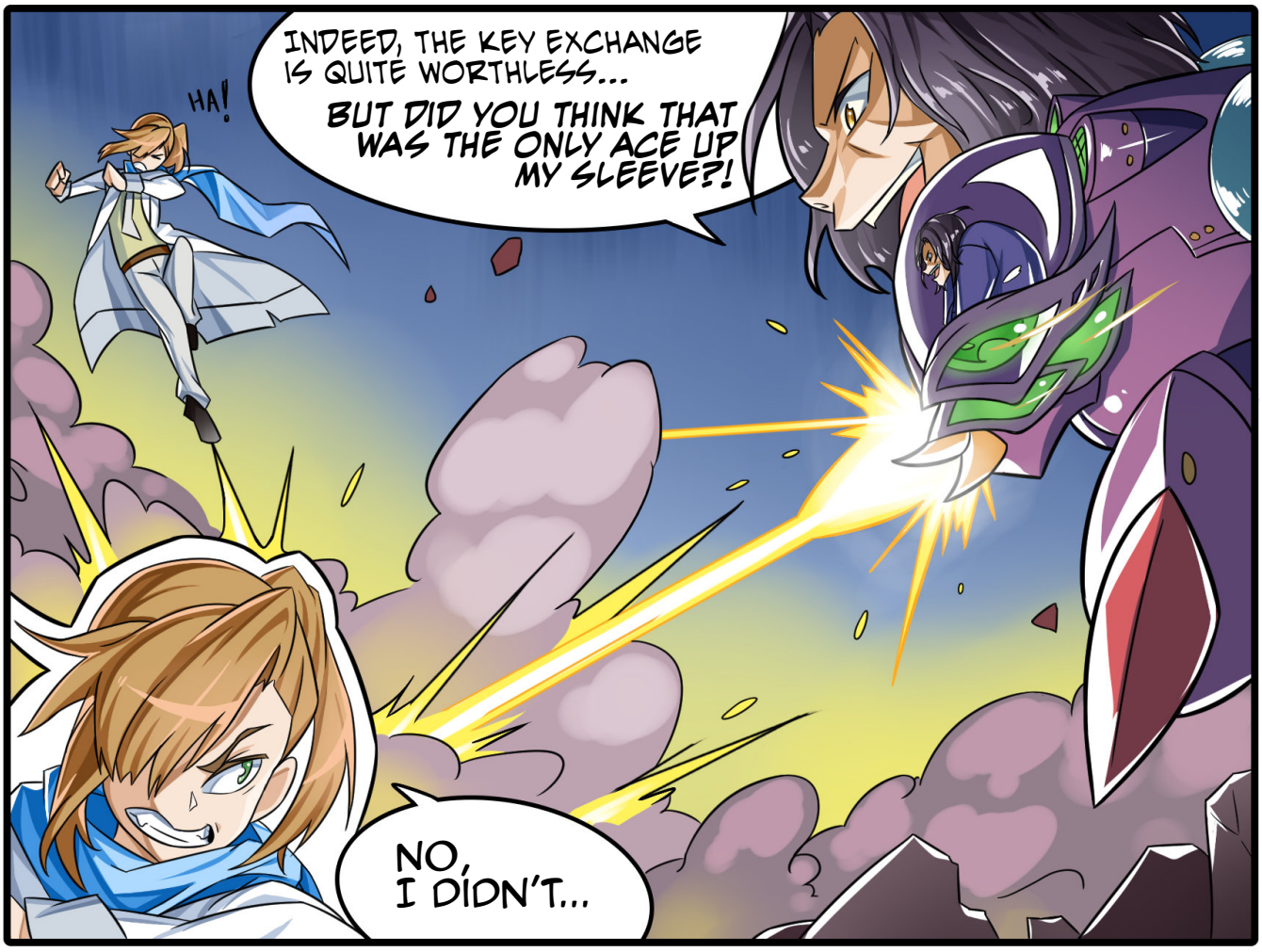

RIIING!
OK, EVERY-ONE!
CLASS DISMISSED!
BY DAY, SHE IS A MATH TEACHER AT VERIFCITY HIGH SCHOOL.
BY NIGHT, SHE IS...
VERIFPAL, THE HERO WHO CAN EXPOSE THE MAYOR'S HIDDEN TYRANNY.
VERIFLAB

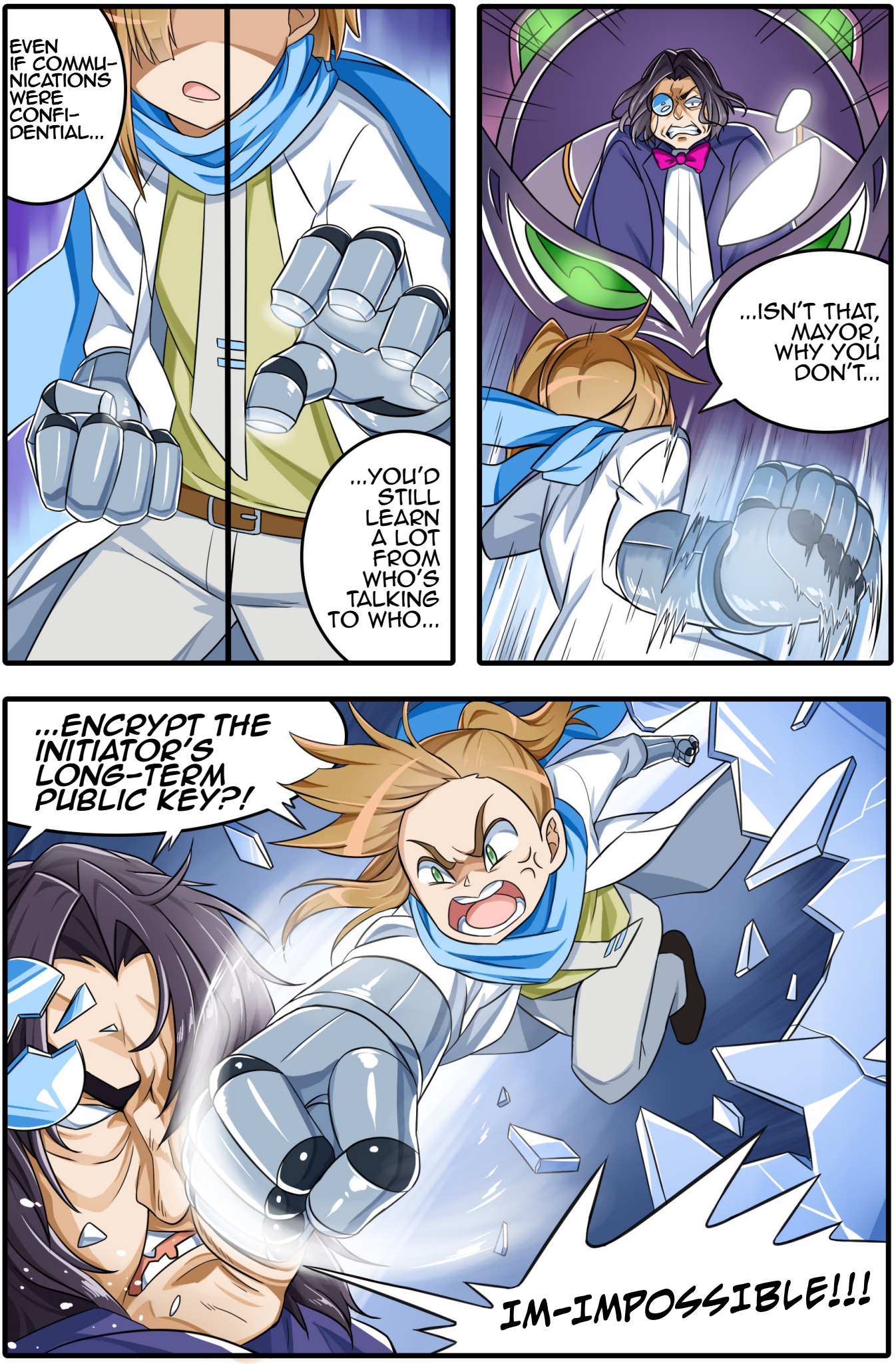

SO...
THIS IS SUPPOSED TO BE THE SECURE PROTOCOL EVERYONE IS USING TO COMMUNICATE...
I'M SURE I'VE MODELED IT CORRECTLY...
ALICE'S EPHEMERAL KEY... IT'S THE ONLY THING KEEPING HER MESSAGES SAFELY ENCRYPTED...
EPHEMERAL KEY
YEARS EARLIER...
NO, VERIFPAL. A COMPROMISED EPHEMERAL KEY CAN STILL MEAN TROUBLE.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

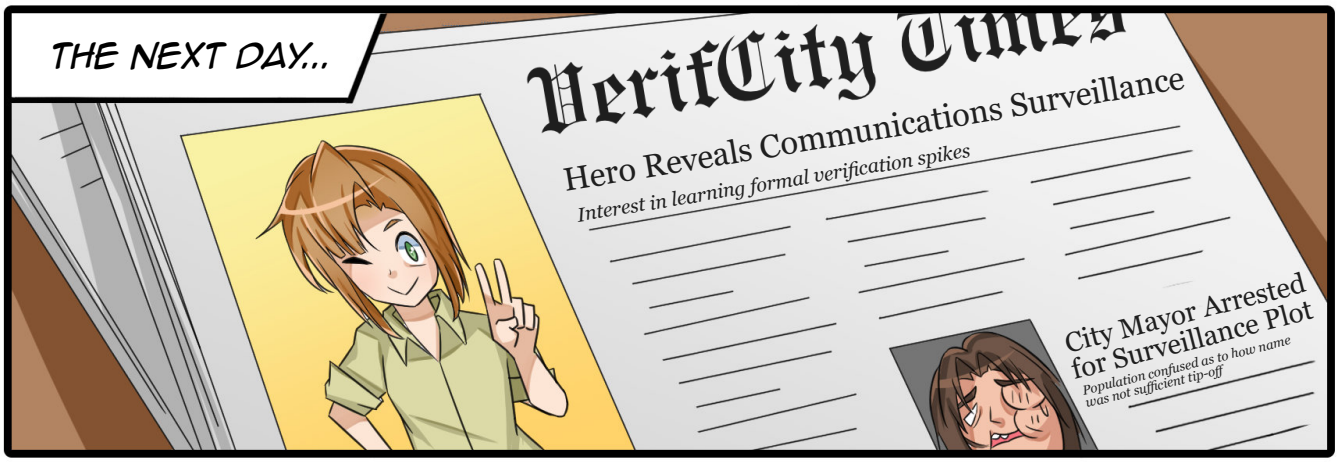

THAT'S RIGHT!
NORMALLY, MESSAGE KEYS ARE DERIVED NOT ONLY FROM ALICE'S EPHEMERAL KEY, BUT ALSO FROM A ROOT KEY...
Alice's Ephemeral Key
Master Secret
HKDF
HKDF
THIS IS WHAT TIES THE MESSAGES TO ALICE AND BOB'S IDENTITIES...
!
HKDF
Root Key
ENCRYPT
BUT... THE ROOT KEY IS NEVER GETTING MIXED INTO ALICE'S NEW MESSAGE ENCRYPTION KEY!

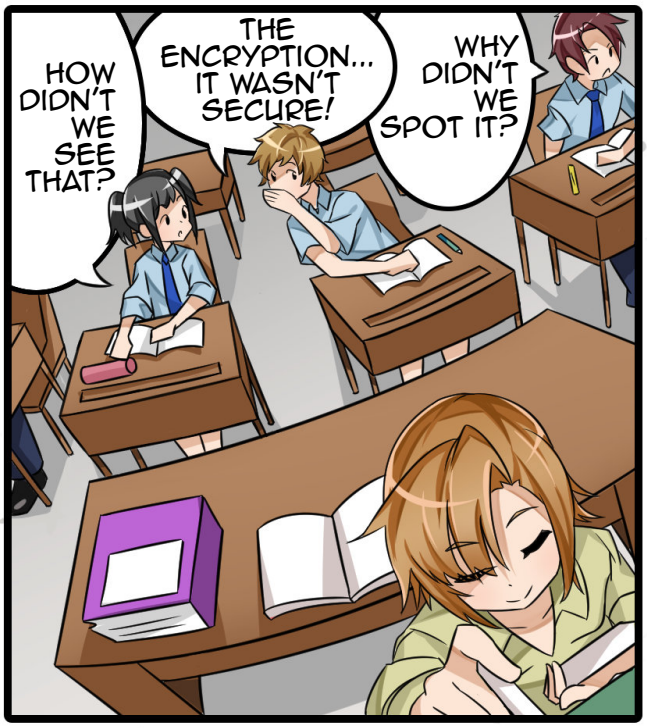

WHICH MEANS...
IF AN ACTIVE ATTACKER REPLACES THE EPHEMERAL KEY,
THE ENTIRE SESSION GETS COMPROMISED!
THEY CAN READ EVERYTHING!
SMASH

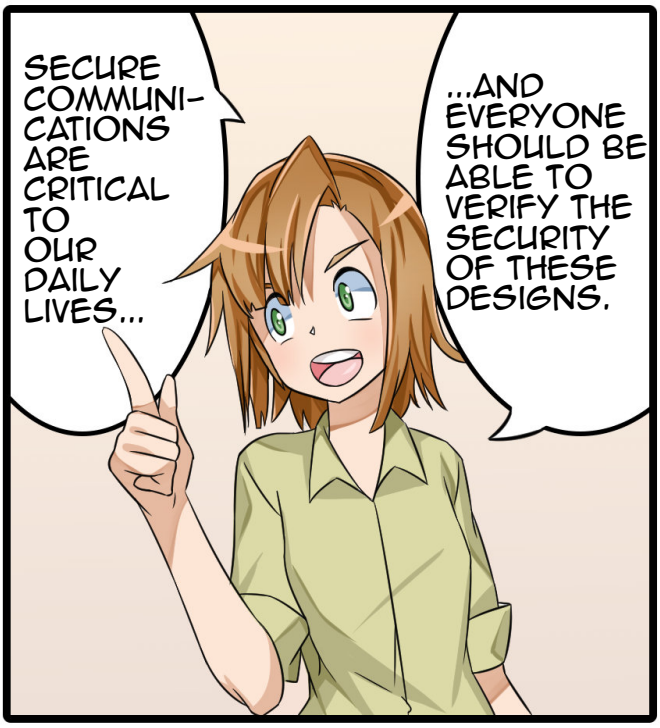

EXCELLENT DEDUCTION AS ALWAYS, VERIFGAL!
MAYOR N. D. MIDDLE!

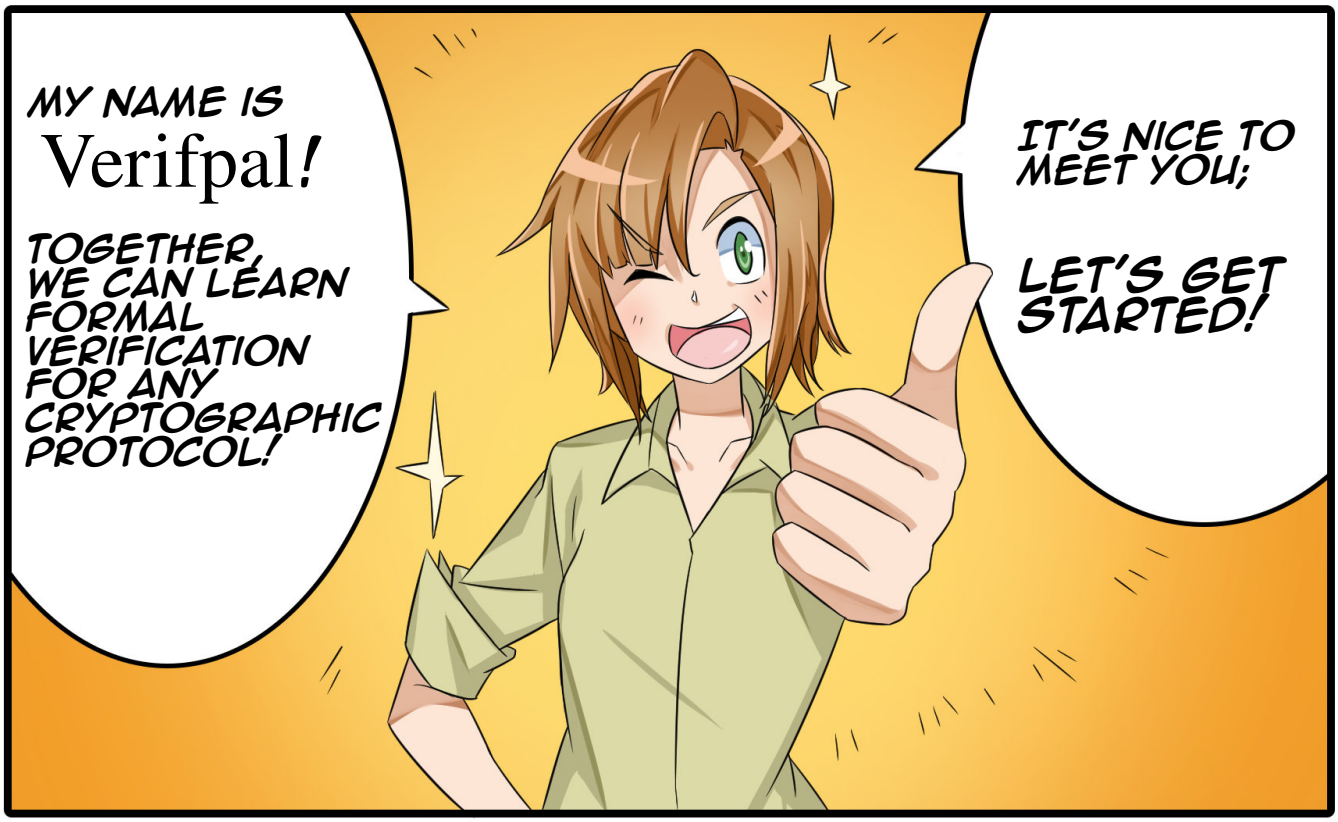

INDEED, THE KEY EXCHANGE IS QUITE WORTHLESS...
BUT DID YOU THINK THAT WAS THE ONLY ACE UP MY SLEEVE?!
HA!
NO, I DIDN'T...


EVEN IF COMMU-NICATIONS WERE CONFI-DENTIAL...
...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

PART I

Using Verifpal

CHAPTER 1

Setting Up Verifpal

This chapter explains how to install Verifpal, run an analysis and use its editor integrations and report formats.

What Verifpal's Assurance Covers

Verifpal searches for attacks; it does not prove their absence. A passing query means only that a bounded search found no attack, and the verdict says how far that search went. A reported attack has been re-executed and checked by a validator that is separate from the search, but two known limitations of the session model can still produce a false report (Chapter 5). For high-assurance work, reconstruct important attacks by hand and compare important results with those from an established tool such as ProVerif [2].

1.1 Installing Verifpal

Verifpal supports Windows, Linux and macOS. The Verifpal website[1] provides installation instructions, and the releases page[2] contains the available builds.

Use a package manager when possible so that Verifpal can be upgraded with the rest of the system.

1.1.1 Windows

On Windows 10 and 11, install Verifpal with winget:

```
winget install SymbolicSoftware.Verifpal
```

Alternatively, use Scoop[3] after adding the Verifpal bucket:

```
scoop bucket add verifpal https://github.com/symbolicsoft/verifpal.git
scoop install verifpal
```

[1] https://verifpal.com
[2] https://github.com/symbolicsoft/verifpal/releases
[3] https://scoop.sh

1.1.2 macOS

On macOS, install the Verifpal cask from the project's Homebrew[4] tap:

```
brew tap verifpal.com/source https://github.com/symbolicsoft/verifpal
brew install --cask verifpal
```

If an older Homebrew formula is already installed, first run `brew uninstall verifpal`. Then run the commands above. Future versions can be installed with `brew upgrade`.

1.1.3 Linux

Each release provides packages for Debian-based, RPM-based, Alpine and Arch Linux distributions. Packages are available for x86-64 and ARM64. Replace `amd64` with `arm64` on an ARM64 system:

```
sudo dpkg -i verifpal_*_linux_amd64.deb
sudo rpm -i verifpal_*_linux_amd64.rpm
sudo apk add --allow-untrusted verifpal_*_linux_amd64.apk
sudo pacman -U verifpal_*_linux_amd64.pkg.tar.zst
```

Use `dpkg` on Debian and Ubuntu; `rpm` on Fedora, RHEL and openSUSE; `apk` on Alpine; and `pacman` on Arch. The Alpine package is unsigned, so its command includes `--allow-untrusted`.

These packages also install manual pages and shell completions for bash, zsh and fish. Run `man verifpal` to open the main manual page.

1.1.4 Release Archives

If no package is suitable, download an archive from the releases page. Each archive contains the `verifpal` binary (`verifpal.exe` on Windows), manual pages, shell completions and example models. Select the archive for the operating system and architecture:

- `darwin_arm64` for Apple silicon or `darwin_amd64` for an Intel Mac;
- `windows_amd64` for 64-bit Windows; or
- `linux_amd64` or `linux_arm64` for Linux.

Each release also includes SHA-256 checksums and a software bill of materials for every archive.

Extract the archive and run the binary from a terminal. To invoke it from any directory on a Unix-like system, copy it to a directory on the system `PATH`:

[4] https://brew.sh

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

A manually installed binary does not update itself or register shell completions. Check the releases page for updates. To install a completion script yourself, ask Verifpal to print it and redirect the output to the appropriate location. For example, on a typical system using bash:

```
verifpal completion bash > /etc/bash_completion.d/verifpal
```

1.1.5 Building from Source

Building Verifpal requires the Rust toolchain.[5] To build and install the published crate on the current user's `PATH`, run:

```
cargo install verifpal
```

To build a cloned repository with Git,[6] run the following command from the repository root. The binary is written to `target/release/`:

```
cargo build --release
```

Both commands build the same source used for official releases, but neither installation updates automatically. A repository clone also provides the `make build`, `make test` and `make lint` targets. Run `make dist-assets` to generate shell completions and manual pages in `target/dist-assets/`; `cargo build` does not generate these files.

Verifpal is free software distributed under the GNU General Public License, version 3. The license text is available at `https://www.gnu.org/licenses/gpl-3.0.en.html`.

1.2 Running Verifpal

Run `verifpal` without arguments to confirm the installation and list the available commands:

[5] `https://www.rust-lang.org/tools/install`
[6] `https://git-scm.com/book/en/v2/Getting-Started-Installing-Git`

• RUNNING VERIFPAL

```
Verifpal 1.4.1 - https://verifpal.com

Usage: verifpal <COMMAND>

Commands:
verify      Analyze protocol models and report attacks against their queries
pretty      Reformat models into the canonical Verifpal style
diagram     Emit a Mermaid sequence diagram of a model
about       Print version, homepage and contributor credits
lsp         Run the Verifpal language server over stdio
completion  Print a shell completion script to stdout
help        Print this message or the help of the given subcommand(s)

Options:
-h, --help     Print help (see more with '--help')
-V, --version  Print version
```

- `verify` analyzes one or more model files with the `.vp` extension. This is the main command used in this manual. A model file's name must end in `.vp` and be at most 64 characters long.
- `pretty` prints the model in Verifpal's canonical format. The `--write` option reformats files in place and lists every file it changed. The `--check` option lists files that require formatting and returns exit status 1 if there is at least one, which is useful in automated checks. Formatting requires only that the model parse; a model that fails validation is still formatted.
- `diagram` converts a model to a Mermaid `sequenceDiagram`. Every message becomes an arrow, guarded values keep their brackets, and everything a principal knows, generates or computes becomes a note over that principal. The model is neither validated nor analyzed.
- `about` prints information about Verifpal and its license.
- `lsp` starts the built-in Language Server Protocol implementation used by editor integrations. It communicates through standard input and output and is normally started by an editor.
- `completion` prints a completion script for `bash`, `zsh`, `fish`, `elvish` or `powershell`. Use it when the installation method did not provide completions.

Every command accepts `--help`. The long form, `verifpal help verify`, documents each option in full.

The `verify` command supports several output and automation options:

- `--format json` emits the complete analysis as one JSON object, including every attack trace as structured steps. This is what the editor integrations consume.
- `--format html` emits a self-contained HTML report (§1.2.6).

- `--format tex` emits a complete LaTeX report (§1.2.7).
- `--fail-on-attack` returns exit status 2 if any query is contradicted. This option allows a continuous-integration job to fail without parsing the output.
- `--quiet` prints only verdicts, attack traces and warnings; `--verbose` adds every value the attacker deduced and the rule that yielded it. The deduction log is capped after a thousand messages on a large model. The two options conflict.
- `--color auto`, `always` or `never` controls terminal color. The default honors the `NO_COLOR`, `TERM=dumb`, `CLICOLOR_FORCE` and `FORCE_COLOR` environment variables.

The JSON, HTML and LaTeX formats write only the requested document to standard output. They omit terminal progress and color. Parse errors appear inside the generated document, so redirection captures the complete result.

Verifpal exits with status 0 when every model was analyzed, whether or not attacks were found. Status 1 means that a model could not be read, parsed or validated; the remaining models are still analyzed. Status 2 is returned only with `--fail-on-attack`, and status 1 takes precedence over it.

The first output line includes the installed version. Include this version when reporting a problem. The `verify` and `about` commands also check in the background whether a newer release exists, and print one line on standard error if so. This check never delays a run, and it is the only network request the tool makes.

1.2.1 Reading the Terminal Output

An analysis prints a banner, progress lines and a final report. Each progress line is prefixed with a label: `Info` for the attacker type, the session count and the current phase; `Fail` when a query is contradicted during the search; and `Warning` for declared assumptions and peer scenarios. The final report lists every query in source order with its verdict, followed by the attack trace of every contradicted query and a summary line.

In a terminal that supports color, the verdict labels are `Pass ✓` and `Fail ×`, and trace lines are drawn with box characters. When color is disabled, the same output uses `PASS +`, `FAIL x`, `Info .`, `Warning !` and plain `|` and `>` markers. The result boxes in this manual use the colored labels with plain trace markers, and they omit the banner and progress lines. They also abbreviate long derived terms by the model's own name for them, where the tool prints the term in full.

A passing verdict carries an *envelope* in brackets that records how far the search went:

```
Pass ✓ confidentiality? m1  [search exhausted at 2 sessions]
```

Exhausted means that the search space Verifpal defines was fully explored at the stated session count. It is not a proof that no attack exists (§5.6). If the search declined a proposal because it exceeded the model's term-depth bound, every passing query in that model is instead marked `[search truncated: term depth]` (§5.3). A contradicted query carries a trace rather than an envelope.

1.2.2 Choosing How Many Sessions to Analyze

By default, Verifpal analyzes two concurrent sessions of every declared principal. A single session cannot represent attacks that require two fresh values from the same role (§5.6.3). Use `--sessions` to select a value from `1` through `16`:

```
verifpal verify --sessions 3 model.vp
```

Most analyses can use the default. Use `--sessions 1` if a large model is too slow; cost grows steeply with the number of replicated principals, and a two-session analysis of a mid-sized model costs several times as much as a single-session analysis. A higher count can find attacks that require three or more sessions, but it also increases the cost.

Each session clones every principal. The 128-principal limit described in Chapter 2 applies after cloning. A model with 40 principals can therefore use at most three sessions. If the requested count exceeds the limit, Verifpal refuses the analysis and names a count that fits.

Traces and results identify replicated principals and values with a `#` suffix. For example, Alice's second session is `Alice#2`, and its copy of `na` is `na#2`. These names are generated by Verifpal and do not appear in the model source. The `Info` line at the start of every analysis states the session count in use.

1.2.3 Testing Verdict Stability Across Session Counts

The `--saturate` option searches for a stable session count. It begins at the default of two sessions and continues to three and then four. It stops at the first count whose result code (§1.2.5) equals the previous count's, and reports the analysis from that count. The ladder starts at the default rather than at one session because an attack that first needs three concurrent runs leaves one and two sessions reading alike; a ladder that began at one would have called that saturated before it looked above the default. The intermediate runs print nothing, and the reported run is not analyzed twice. An `Info` line states the outcome:

```
Verdicts were unchanged from 2 sessions through 3; analyzing at 3.
```

If the verdicts were still changing at four sessions, the line says so instead. This result is empirical, not a proof. It states only that the verdicts did not change over the counts examined. Because the ladder stops at the first repeated code, an attack that first requires four runs is missed when two and three sessions read alike, and nothing above four is examined. A claim about a particular count therefore requires an explicit `--sessions` run rather than an inference from `--saturate`. No finite session count removes the bound of §5.6.3.

If a verdict changes between one and two sessions, at least one property depends on concurrent runs of a role. Examine the corresponding attack trace to understand that dependency.

Adding sessions gives the attacker more material. An attack found at a lower count must not disappear at a higher count. If it does, Verifpal warns on standard error that the result indicates an analysis error rather than a protocol property.

1.2.4 Generating Exploratory Queries

An analysis covers only the queries in the model. The `--auto-queries` option temporarily replaces the model's **`queries`** block with a generated set:

```
verifpal verify model.vp --auto-queries
```

Verifpal generates the queries according to three rules:

- It asks for the confidentiality of every generated value and every value declared with **`knows private`**.
- It asks whether the recipient authenticates the declared sender for each transmitted value that the recipient later uses in a primitive.
- It asks whether each received value used in a primitive is fresh.

Verifpal does not generate unlinkability or equivalence queries because each requires a meaningful pair of values chosen by the model author. It does not generate query options either.

The option does not modify the file, and the model's own **`queries`** block must still be present and valid because the model is validated before its queries are replaced. It may produce many unsurprising failures: a long-term public key is not fresh, for example, and a deliberately published value is not confidential. Its purpose is to reveal failures that the original query set omitted.

Use generated queries to explore a model, not as a substitute for a deliberate specification. A model submitted for review should contain a curated **`queries`** block whose entries correspond to documented security goals. If a generated query reveals a relevant failure, add that query explicitly and document why it matters.

1.2.5 The Result Code

The `--result-code` option appends a compact, machine-readable summary of all query verdicts to the analysis and suppresses the banner.

The result contains one letter–digit pair per query, in source order. The letter identifies the query type: `c` for confidentiality, `a` for authentication, `f` for freshness, `u` for unlinkability and `e` for equivalence. The digit is `0` when no attack was found and `1` when the query was contradicted. A confidentiality query followed by a contradicted authentication query produces:

```
c0a1
```

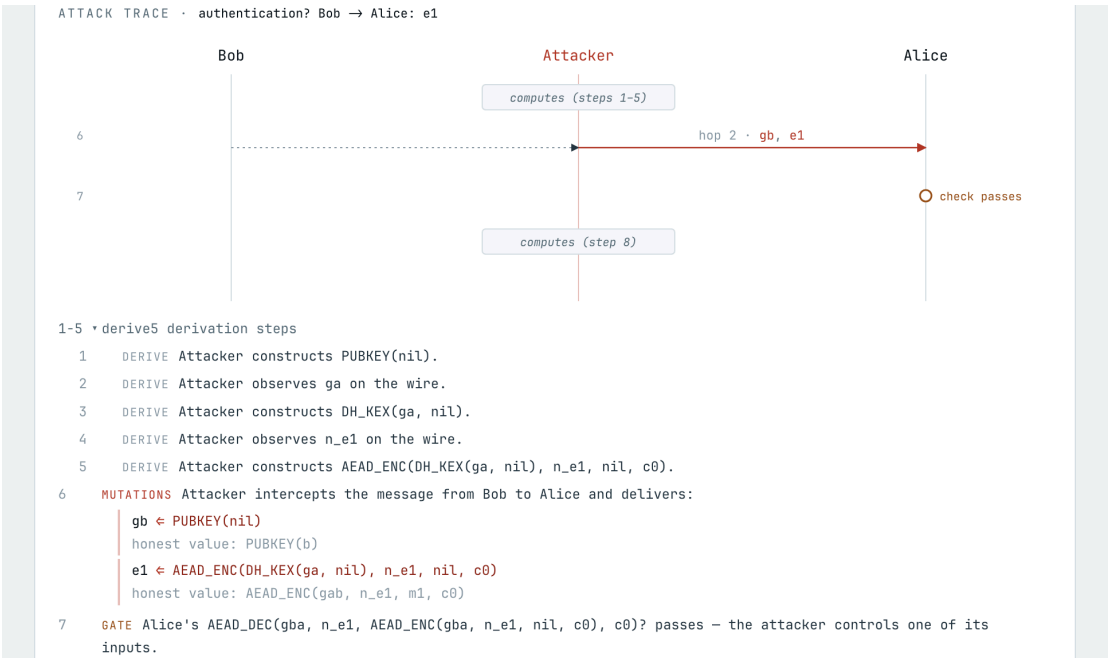


FIGURE 1.1 An HTML report for a man-in-the-middle attack on an unauthenticated Diffie–Hellman exchange. The diagram and text use the same step numbers. The *computes* box contains attacker derivations, the solid red arrow marks a forged delivery, and each replacement appears beside the honest value it replaced.

Because the code is the final output line, a script can extract it with `tail -1` and compare it with an expected value. Adding `--quiet` silences everything but the code. When several models are given, each code is prefixed with the path of its model. Verifpal's own test suite pins the expected verdict of every example model this way, and it is the easiest way to compare two runs.

1.2.6 The HTML Report

Use `--format html` to render the complete analysis as a self-contained web page:

```
verifpal verify model.vp --format html > report.html
```

The generated file embeds its styles, scripts, diagrams and syntax highlighting. It can be read offline without JavaScript and can be printed or shared directly. If the command receives several models, the report contains one section for each model.

Each section begins with a diagram of the original protocol. It numbers messages in transmission order, retains brackets around guarded values, draws phases as horizontal rules and marks each **`leaks`** statement on the relevant principal's lifeline. The values a principal generates and computes between messages appear as notes on its lifeline, so the diagram shows the local computation as well as the wire. Values replaced or replayed in a reported attack appear in red. A query table follows, with source locations, verdicts, the envelope of every passing verdict and conclusions for contradicted queries.

Each attack appears as both a diagram and a numbered trace of the same reduced witness (§5.4). The diagram adds an *Attacker* lane to the participating principals. A capture followed by a forged delivery represents interception; dashed arrows represent replays; and labels on principal lifelines record passed or defeated checks. Consecutive derivations are grouped in a *computes* box. The text trace lists each replacement as the delivered value followed by the honest value it displaced. Diagram rows and trace entries share step numbers, and hovering over either highlights the corresponding element.

The report ends with syntax-highlighted model source. Each query is colored by verdict and linked from the query table. The report uses the same lexer as the analysis engine.

1.2.7 The LaTeX Report

Use `--format tex` to create a printable LaTeX report. The report typesets protocol terms as mathematics and contains reusable figures. `--format latex` is an alias for the same format.

```
verifpal verify model.vp --format tex > report.tex
```

The generated file is a complete LaTeX document with its own preamble. Compile it with `tectonic report.tex`. A report for several models begins with a summary table that lists each model's attacker, session count, result code and number of attacks.

Each model receives its own section. A message-sequence chart shows a lane for every principal; numbered messages; local generations and computations; phases; leaks; and guarded values. A dagger marks any value substituted or replayed by an attack in the report. The query table records each query's source line, verdict and scope. For a passing query, the scope states where the search ended, such as *search exhausted at 2 sessions*.

For each contradicted query, a figure and a numbered trace describe the same reduced witness (Figure 1.2; §5.4). The figure shows participating principals, an *Attacker* lane, forged deliveries, replays, checks and grouped derivations. Its row numbers correspond to the text trace below. A scope note records the attacker and session count, along with any relevant weakening assumptions (§4.6), peer scenarios (§2.7) or incomplete searches.

The report includes the syntax-highlighted, line-numbered model source as an appendix. Query-table links refer to these line numbers. Its final section also states that a passing query is not a proof that no attack exists.

Primitive names are upright, and text after an underscore becomes a subscript. The `?` on a checked primitive becomes a superscript. Constants retain their source names. If a term cannot be parsed for mathematical typesetting, the report displays it as escaped upright text.

Comment markers delimit every generated figure, table, listing and preamble block: `%% --- BEGIN verifpal` and `%% --- END verifpal`. Figures depend only on the `\vp`

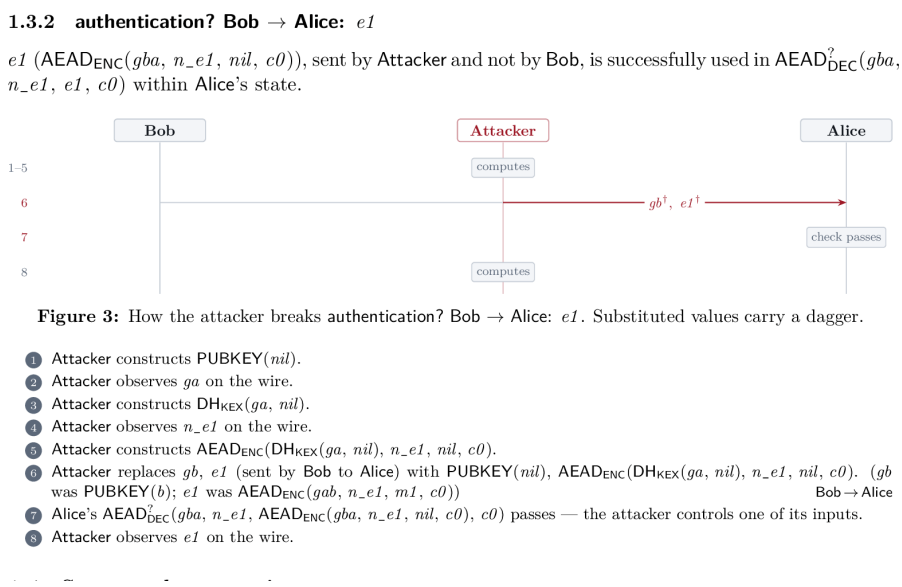


FIGURE 1.2 A LaTeX report for the authentication attack in Figure 1.1. The attacker occupies the center lane, derivations are grouped in a *computes* box, and margin numbers correspond to the trace below. Protocol terms are typeset mathematically: `AEAD_ENC` becomes $\mathsf{AEAD}_{\mathsf{ENC}}$, a checked primitive uses a superscript question mark, and a substituted value carries a dagger.

macros in the marked preamble block. To reuse a figure, copy that preamble block once and then copy the figure. LaTeX lays out and scales each diagram for the available line width. Attack figures and tables do not float, which keeps them next to their explanatory traces.

Use HTML for interactive reading and sharing. Use LaTeX for a PDF, a report appendix or reusable publication figures.

1.3 Editor Integrations

Each editor integration uses the language server built into the Verifpal binary. Editor diagnostics therefore come from the same parser and analyzer as command-line results. Install the Verifpal binary before installing an editor extension.

1.3.1 Visual Studio Code

The Visual Studio Code extension[7] is available from the editor's extension panel under *Verifpal*. It reports parse and sanity errors in the Problems panel while a model is edited. Hover information documents primitives, constants, queries and keywords. Primitive documentation includes argument and output counts, support for `?`, and available weakening assumptions (§4.6). Constant documentation shows its assigned term, origin, recipients and phases. The extension also provides completion, signature help, definition and reference navigation, rename, folding, a model outline and canonical formatting.

[7] Visual Studio Code is available at `https://code.visualstudio.com/`. The extension requires Verifpal 1.1 or later and Visual Studio Code 1.82 or later.

Start an analysis with *Verifpal: Run Attacker Analysis* from the Command Palette, the editor title bar or the action above the **`queries`** block. The language server runs the analysis in the background, and the progress notification can cancel it. Contradicted queries and their traces appear in the Problems panel; complete output appears in the *Verifpal Analysis* pane. The *Verifpal: Show Protocol Diagram* command draws a diagram and refreshes it whenever the model is saved.

The extension exposes the following settings:

- `verifpal.enabled` enables or disables the extension;
- `verifpal.path` selects the Verifpal binary;
- `verifpal.validateOnType` controls validation while typing;
- `verifpal.analyzeOnSave` runs analysis when a model is saved;
- `verifpal.sessions` sets the session count (§1.2.2);
- `verifpal.diagnostics.passing` controls diagnostics for passing queries;
- `verifpal.inlayHints.argumentNames` controls argument-name hints; and
- `verifpal.codeLens` controls code lenses.

1.3.2 Neovim

Install the Neovim plugin[8] with a plugin manager. It provides highlighting, hover information, completion, signature help, diagnostics, folding, symbols, navigation, rename and formatting through the language server.

The plugin provides the following commands:

- `:VerifpalVerify` analyzes the current buffer, including unsaved changes, in the background and attaches each verdict to its query.
- `:VerifpalCancel` stops the current analysis.
- `:VerifpalResults` displays conclusions and numbered attack traces.
- `:VerifpalDiagram` displays the protocol; `:VerifpalDiagram!` displays its Mermaid source.
- `:checkhealth verifpal` reports the detected binary and supported features.

1.3.3 Zed

Install the Zed extension[9] by searching for *Verifpal* on the editor's extensions page. A tree-sitter grammar provides immediate highlighting. To use parser-supplied semantic highlighting instead, set `semantic_tokens` to `combined` for Verifpal. The language server supplies diagnostics, hover information, completion, signature help, navigation, rename, inlay hints, a model outline and formatting.

Start an analysis from the code lens above the **`queries`** block. Because Zed disables code lenses by default, first set `code_lens` to `on`. The language server runs in the background

[8] The plugin requires Verifpal 1.1 or later and Neovim 0.12.4 or later. Point a plugin manager at the `verifpal-nvim` repository linked from the Verifpal website. Calling `setup()` is optional.

[9] Zed is available at `https://zed.dev`. The extension requires Verifpal 1.1 or later and Zed 1.16 or later.

and attaches each verdict to its query: an error for a contradiction and a hint otherwise. Zed tasks also provide default- and single-session verification, formatting, a formatting check and a Mermaid diagram. Run a task with `task: spawn` from a `.vp` buffer. Tasks save the buffer before invoking the command-line tool.

1.4 The Verifpal Workbench

The Verifpal Workbench runs Verifpal in a browser through WebAssembly. It can edit, analyze and share models without a local installation, and it uses the same analysis entry point as the command-line tool, so the two cannot disagree about what a model means. Open the Workbench.

CHAPTER 2

The Verifpal Modeling Language

This chapter introduces the structure and syntax of a Verifpal model through a small protocol.

Every model has four required elements. It first selects an *attacker*. It then interleaves blocks of local operations by *principals* with the *messages* they exchange over the network. A final *queries* block states the security properties to analyze. A model may also contain phases and multi-peer scenarios. The attacker declaration must come first, and the queries block must come last.

The following sections explain each part and then introduce constants, phases and multi-peer scenarios.

2.1 A First Protocol

Figure 2.1 shows the protocol portion of a model. Alice and Bob exchange unauthenticated public keys and derive a shared secret. Bob then uses the secret to encrypt a message for Alice. The rest of this section explains the protocol in execution order; §2.8 adds the required queries block.

Alice first generates the fresh private value `a`. She derives its public key, `ga = PUBKEY(a)`, which represents g^a in a Diffie–Hellman group, and sends `ga` to Bob.

Bob holds the private message `m1`. He generates his own private value `b` and public key `gb`. From Alice's public key and his private value, he computes `ss_a = DH_KEX(ga, b)`, representing the shared secret g^{ab}. He also generates `n1`, the nonce for the authenticated encryption; a nonce is an ordinary generated constant, and each encryption under one key needs its own (§4.4). Bob encrypts `m1` under the shared secret and sends Alice `gb`, the nonce and the ciphertext `e1`.

Alice computes `ss_b = DH_KEX(gb, a)`. The Diffie–Hellman equation makes `ss_a` and `ss_b` equivalent. She then attempts to decrypt `e1` with `AEAD_DEC`, which needs the same key and the same nonce.

The question mark on Alice's last operation makes the decryption a *checked primitive*. If the check fails, Alice stops and performs no later operation. Section 4.5 defines checked primitives.

Chapter 4 documents the remaining primitives, including hashes, signatures, key encapsulation and secret sharing.

• SIMPLE PROTOCOL

```
attacker[active]
principal Alice[
    generates a
    ga = PUBKEY(a)
  ]
Alice → Bob: ga
principal Bob[
    knows private m1
    generates b, n1
    gb = PUBKEY(b)
    ss_a = DH_KEX(ga, b)
    e1 = AEAD_ENC(ss_a, n1, m1, gb)
  ]
Bob → Alice: gb, n1, e1
principal Alice[
    ss_b = DH_KEX(gb, a)
    e1_dec = AEAD_DEC(ss_b, n1, e1,
         gb)?
  ]
```

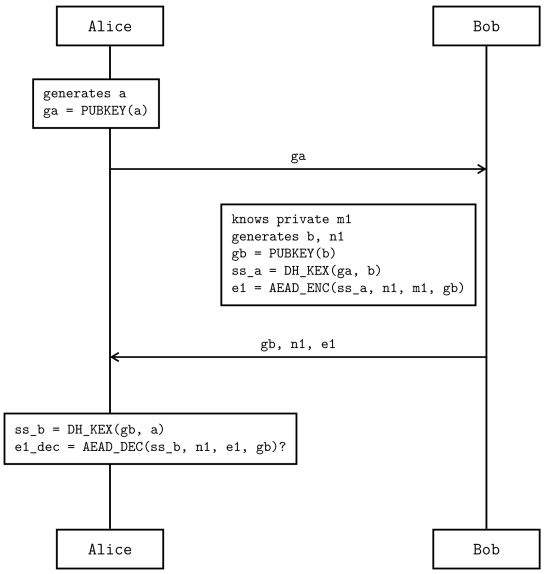


FIGURE 2.1 The protocol actions of a Verifpal model and their message-sequence diagram. The diagram is explanatory and is not part of the model source.

2.2 Declaring the Attacker

The first line selects the attacker. Use **attacker**[passive] or **attacker**[active].

A passive attacker observes every value sent over the network and derives any value allowed by the symbolic equations, but it cannot change traffic. An active attacker can also intercept, replace and inject values. Chapter 3 defines both attacker models in detail.

2.3 Principals

A principal declaration consists of a name and a block of local operations. In the following example, Alice knows the public constants `c0` and `c1`, knows the private message `m1`, and generates a fresh value `a`. The attacker initially knows the public constants but not `m1`.

• NEW PRINCIPAL: ALICE

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

Bob can be declared in the same way and can perform computations in the declaration block:

• NEW PRINCIPAL: BOB

```
principal Bob[
    knows public c0, c1
    knows private m2
    generates b
    gb = PUBKEY(b)
  ]
```

A principal may have several declaration blocks. Each later block continues the principal's state from its previous block. This allows a model to alternate local operations and network messages in protocol order, as Figure 2.1 does for Alice.

A model may declare at most 128 principals. Principal names are case-insensitive and normalized to title case, so `alice` and `Alice` refer to the same principal.

2.4 Constants

In the preceding examples, `c0, c1, m1, m2, b, gb` are *constants*. Every Verifpal term is either a constant or a primitive applied to other terms. Constants follow four rules:

- *Immutability.* Once assigned, constants cannot be reassigned.
- *Global namespace.* A constant name denotes one value throughout the model. Several principals may repeat the same **`knows`** declaration, with the same qualifier, to share prior knowledge of that value. A **`generates`** declaration or an assignment must introduce a new name; **`leaks`** instead names an existing value.
- *Primitive assignments.* The right-hand side of an assignment must be a primitive expression, not another constant.
- *Reserved names.* A constant may not be named after a Verifpal keyword or primitive, and may not begin with **`attacker`** or `unnamed`, since Verifpal uses those prefixes internally.

These rules ensure that each constant denotes one value throughout the model.

All identifiers are case-insensitive. Verifpal normalizes identifiers other than principal names to lowercase. Thus, `aead_enc` and `AEAD_ENC` name the same primitive, while `M1` and `m1` name the same constant.

Use a single underscore (_) on the left of an assignment when its result will not be referenced. This is useful for a checked primitive such as `_ = SIGNVERIF(gb, m, sig)?`, whose output is irrelevant after the check. Each underscore creates a distinct anonymous constant and may be used repeatedly.

Constants enter a model in four ways:

- **`knows`** declares prior knowledge. The **`public`** qualifier gives the value to every principal and to the attacker. A **`private`** value is known only to the principals that declare

it; repeating the same declaration lets several principals share a pre-established secret. Such values represent long-term material and are shared by every session of each declaring principal (§5.6.3). This sharing allows a value created in one session to be accepted in another.

- **`generates`** creates a fresh value, such as an ephemeral private key or an `AEAD_ENC` nonce. Each session of a principal receives a distinct copy (§5.6.3). Re-executing a principal within one session reuses that session's generated values rather than drawing new ones.
- **`leaks`** gives the attacker a constant already known to the principal at that point in the protocol.
- An *assignment* binds a new constant to a primitive expression.

2.5 Messages

A network message appears outside principal blocks and may contain only constants:

• EXAMPLE: MESSAGES

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

The ASCII arrow `→` and Unicode arrow → (U+2192) are equivalent. The `pretty` command normalizes their representation.

In the first message, Alice sends Bob her public key `ga = PUBKEY(a)`. An active attacker may replace this value. In the second message, brackets make `gb` a *guarded constant*. A guard models a value that the recipient has authenticated before the modeled exchange.[1]

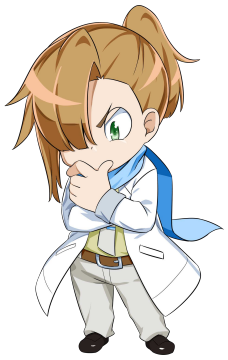
A guard does not hide a constant. The attacker can read `gb`, but cannot change the copy delivered to Alice.

An exception prevents a guard from authenticating attacker-controlled input. If a principal first receives a constant without a guard and later forwards that same constant with a guard, Verifpal still treats the value as substitutable. The forwarding principal had no authenticated basis for the guarded value. The exception covers one forwarding hop only; a value forwarded over two guarded hops is treated as authentic at the second (§5.6.3).

2.6 Phases

Phases divide a model into ordered periods. They support temporal properties such as forward secrecy and post-compromise security under either attacker model. Declare a new phase as follows:

[1] For a public key, *pre-authentication* means that the recipient confirmed the expected key before the protocol session. This prevents an active attacker from substituting its own key in a man-in-the-middle attack.



Guarding the Right Constants

A guard is an assumption about the deployment, not a cryptographic operation performed by Verifpal. Guarding every public key asserts that every recipient authenticated the corresponding key out of band. Guard only values for which the real system provides that assurance.

EXAMPLE: PHASES

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

phase[1]

principal Alice[leaks a2]
```

The initial portion of a model is phase 0. Explicit phase numbers must then increase by one: `phase[1]`, `phase[2]`, and so on. Three rules define their meaning:

- *Knowledge.* The attacker learns a value in the earliest phase in which it is communicated. In the example, it learns `b1` in phase 0 and `a2` in phase 1.
- *Substitution.* The attacker may substitute only a term it could construct in the earliest phase in which the target value is communicated. If the same value is sent in several phases, the earliest transmission determines the available knowledge. Thus, `a2`, leaked in phase 1, cannot help the attacker replace `b1` in phase 0.
- *No retroactive actions.* Knowledge acquired in a later phase cannot justify an earlier action. A key leaked in phase 1, for example, cannot forge a signature checked in phase 0.

A leak in a later phase models a compromise after earlier protocol actions have completed. The attacker may use the leaked material from that phase onward, but not in earlier phases. Chapter 6 uses this construction to analyze forward secrecy. Chapter 8 uses it to model an attacker who records traffic before acquiring a future decryption capability.

A phase may also mark the point at which a cryptographic assumption fails. Section 4.6 introduces per-call-site weakening annotations and the `from phase 2` qualifier.

2.7 Scenarios

The models above use a fixed set of peers. In a deployed system, one party may run the same protocol concurrently with several peers, including a malicious but authorized participant that owns valid long-term keys. A secure protocol should prevent such a participant from using one run to compromise another run with an honest peer.

A `scenarios` block represents one principal running concurrently with different peers under a shared attacker.

2.7.1 A Two-Peer Attack

The Needham–Schroeder public-key protocol provides a standard example of an attack that requires two peers.

The initiator encrypts a nonce and its identity under the responder's public key. The responder returns the initiator's nonce together with a new nonce. The initiator then returns the responder's nonce. Needham and Schroeder published the protocol in 1978; Lowe identified an attack in 1995. The attack combines two concurrent runs rather than breaking either run in isolation.

Alice begins one run with Mallory, who is a legitimate participant with its own key. Mallory re-encrypts Alice's nonce and identity under Bob's key and starts a second run with Bob. Bob accepts the message because both the identity and nonce originated with Alice. He replies with Alice's nonce and a new nonce. Mallory relays this response to Alice in the first run. Alice accepts it because it contains her nonce, then encrypts Bob's nonce under Mallory's key. Mallory decrypts the result. Bob completes the run believing that he communicated with Alice, while Mallory learns the nonce that Bob intended to share only with her.

The attack neither forges a message nor breaks a key. It redirects authentic messages between runs. Lowe's repair includes the responder's identity in the second message. Alice can then detect that a response naming Bob arrived in her run with Mallory.

2.7.2 Declaring Scenarios

A model may declare at most one `scenarios` block. It appears directly between the last principal or message and the **`queries`** block. Each entry names one principal and binds one or more of its constants:

• EXAMPLE: SCENARIOS

```
principal Alice[
    knows private a
    knows public gpeer
    generates ni
    ga = PUBKEY(a)
    m1 = PKE_ENC(gpeer, CONCAT(ni, ga))
  ]

Alice → Bob: m1

scenarios[
    Alice[gpeer = gb]
    Alice[gpeer = gm]
  ]
```

Here, `gpeer` is a placeholder for the peer's public key. The block creates two runs for Alice: one binds `gpeer` to Bob's key `gb`, and the other binds it to Mallory's key `gm`. The placeholder uses **`knows public`** so that the unexpanded model is well formed. Before analysis, scenario expansion removes that declaration and substitutes the bound value.

Separate multiple bindings in one scenario with commas. Each entry can vary constants for only one principal; Verifpal cannot vary two principals within a single scenario entry. Verifpal rejects, at its source position, a scenario that names an undeclared principal, binds a constant the principal does not declare with **`knows`**, refers to a value no principal declares, binds one constant twice, or binds a constant that travels over the wire. A silently ignored binding would analyze the unmodified model and could hide an attack.

2.7.3 Scenario Expansion

Before analysis, Verifpal expands the model in a process similar to session replication (§5.6.3). It clones every principal and message for each scenario and substitutes that scenario's bindings. All expanded runs share one attacker, which can carry information from one run to another. A flight that one scenario's run of Alice sent to its own run of Bob, delivered by the attacker to another scenario's Bob, is a replay in the sense of §3.2.2: Alice sent it once, so it is reported as a duplicate acceptance rather than as a forgery, unless a matching run of Alice emits it.

Scenario expansion follows the same sharing rules as session replication. A generated value, and every value derived from it, is distinct in each scenario. Alice's nonce in the run with Mallory therefore differs from her nonce in the run with Bob. Values declared with **`knows`** remain shared because they represent long-term state. This makes Bob in both scenarios the same identity rather than two unrelated principals.

Traces mark scenario-specific names with an · suffix. For example, the second scenario contains `Alice·2`, `Bob·2` and `ni·2`. These generated names never appear in source. A name may contain both · for its scenario and # for its session.

Before expansion, Verifpal orders the scenarios so that the configurations that stay honest longest come first (§2.7.4); the order written in the model has no meaning. The written query keeps the identifiers of the first scenario in that order. Every other scenario that is honest at phase 0 receives a variant of each query, and a violation in any variant contradicts the written query. A scenario that is corrupt from the start contributes executions and attacker knowledge but no claim of its own.

2.7.4 Honest and Corrupt Counterparties

Verifpal classifies a scenario as corrupt if any bound value depends on a secret constant named by **`leaks`**: a generated value or one declared **`knows private`**. It follows assignments transitively. If `gm = PUBKEY(mk)` and the model declares **`leaks`** `mk`, for example, a scenario bound to `gm` is corrupt. Leaking a public value does not make a scenario corrupt.

Corruption is indexed by phase. A scenario is corrupt from the earliest phase in which a bound value's secret leaks, and honest before it. A peer whose key leaks in **`phase`**`[1]` is therefore an honest peer during phase 0, and phase 0 claims in its run are recorded. Only runs that are honest at the current phase may record a query violation, although the attacker keeps whatever it learns from a corrupt run. If no scenario is honest at the current phase, Verifpal evaluates the queries in the corrupt-peer runs instead, so that the analysis does not report a vacuous pass; the result then includes disclosures that follow from every peer being corrupt. Every result lists the active scenarios and marks each as an honest or a corrupt peer.

A failed checked primitive in an honest execution normally indicates a modeling error: the modeled protocol cannot complete even without attacker interference. Verifpal therefore rejects it. A run with a corrupt peer may legitimately stop. For example, Bob should not be able to decrypt a message intended for Mallory. Verifpal requires successful checks in honest scenarios but permits a corrupt scenario to halt.

All other analysis rules remain unchanged, and the attacker retains anything it learns from a corrupt run.

2.7.5 Cost and Limits

Scenario and session counts multiply. Two scenarios at the default of two sessions create four copies of each principal. The 128-principal limit applies after both expansions. Verifpal also limits the combined expansion to 31 copies and reports a valid session count if the requested combination is too large.

Use scenarios for properties that depend on peer identity, such as identity misbinding, unknown-key-share attacks, impersonation or attacks that require one party to communicate with two peers concurrently. A fixed set of principals is simpler and less expensive when the property does not depend on this distinction.

Scenarios have two important bounds. First, each scenario binds constants for one named principal; it does not quantify a role over an arbitrary population. Second, anal-

ysis covers only the declared number of peers. A model with two peers cannot expose an attack that requires three concurrent peers.

2.8 The Queries Block

Every model must end with a **queries** block that states its security properties. Any declaration after this block is an error. This rule prevents trailing protocol actions from being omitted from the analysis.

For Figure 2.1, the queries ask whether the attacker can obtain the ciphertext or plaintext, whether Alice authenticates Bob as the source of the ciphertext, and whether Alice and Bob derive equivalent shared secrets:

• EXAMPLE: QUERIES

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

Under a passive attacker, only the first query is contradicted because `e1` is sent over the network. The plaintext remains secret, the source is not altered and both shared-secret terms are equivalent. Under an active attacker, all four queries are contradicted because the public keys are unauthenticated. Chapter 3 explains each query and the corresponding attacks.

2.9 Comments

Use `//` to comment out the remainder of a line. Block comments begin with `/*` and end with `*/`; they may span several lines.

The `pretty` command preserves every comment. Four positions cannot hold a comment again unambiguously: inside a primitive argument list, inside the brackets of **attacker**`[...]` or **phase**`[...]`, inside a scenario's binding brackets, and inside a query option's inner brackets. A comment written there is moved to the line above the statement that contains it.

CHAPTER 3

Queries and the Attacker

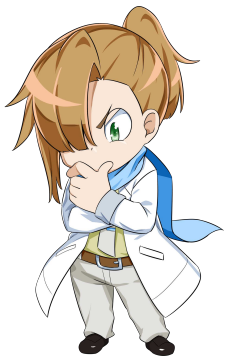
A model describes a protocol; a query states a property that the protocol should satisfy. Verifpal searches for an attacker-controlled execution that contradicts each query. This chapter defines the five query types, the attacker models and the interpretation of results.

A useful query follows from a defined threat model and a specific security goal. Signal, for example, aims to protect earlier messages after a device compromise. A protocol model is informative only when its queries state such goals precisely.

3.1 Use Cases and Security Goals

Begin by defining the protocol's *use case*: the participants, the data they exchange and the environment in which they operate. A protocol for a phone call may require different properties from one connecting an ATM to its bank.

Next, identify the *principals* and the *security goals* promised to each one. A messaging model may contain two users or an entire group; an HTTPS model contains a client and a server. State each promise independently. Confidentiality does not imply authentication: encrypted data may remain confidential while a participant communicates with an impersonator, or authentication may fail in a way that also exposes the data. Query both properties when the use case requires both.



Modeling Private Values

A `knows private` declaration is an assumption about the deployment. A key embedded in a smartphone, for example, may be extractable through hardware analysis. Mark a value private only when the threat model justifies that assumption. Otherwise, the model may overstate the protocol's security.

Message ordering can also affect a security goal. In a group chat, a participant may send several messages before receiving fresh key material, or a new member may join after earlier messages were sent. These executions may have different forward-secrecy and access-control properties.

EXAMPLE QUERIES

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

FIGURE 3.1 Example confidentiality and authentication queries.

A Verifpal model fixes one protocol ordering. To study order-dependent properties, create separate models for the relevant orderings and apply the same queries to each. Compare the resulting verdicts and traces.

3.2 Queries

Queries translate the protocol's security goals into conditions that Verifpal can evaluate.

Verifpal supports five query types. Under a passive attacker, it evaluates the honest execution of each phase. Under an active attacker, it also evaluates every mutated execution validated by the search (§3.3). One counterexample contradicts a query. A passing verdict means that no explored execution contradicted it. The following definitions specify the contradiction condition for each query type; Figure 3.1 shows the first two. The query names are short labels for these conditions and do not denote every standard property of the same name. Where the definition differs from the textbook property, this chapter says so.

3.2.1 Confidentiality Queries

A confidentiality query asks whether the attacker can obtain a value. In Figure 3.1, `m1` is the sensitive value.

> `confidentiality? v` is **contradicted** if and only if, in some explored execution, a principal's state reaches the slot of `v`, and the attacker's knowledge at the end of the deduction fixed point (§5.2.1) contains a value equivalent to the value that `v` resolves to in that state.

Here, *resolves to* means the term obtained after inlining the constant's assignment, after any attacker substitution and after rewriting. If `v = HASH(a, b)`, the query concerns the hash term rather than the name `v`. Equivalence is defined by the equational theory in §4.8. For example, an attacker holding `DH_KEX(PUBKEY(b), a)` also holds the equivalent term `DH_KEX(PUBKEY(a), b)`.

The test concerns the value that the slot holds in the explored execution, not the honest secret known by that name. If the attacker replaces a ciphertext with an encryption of `nil`, the recipient's decrypted slot resolves to the public value `nil`, and a confidentiality

query on that slot is contradicted even though the honest plaintext remains secret. Name the honest secret when that is the intended claim.

A passive attacker may recover `m1` if it obtains the encryption key or all components needed to derive that key. An active attacker may instead replace an unauthenticated public key before Alice derives the key used to encrypt `m1`.

3.2.2 Authentication Queries

An authentication query asks whether every value that a recipient accepts and uses originated with the claimed sender, and whether each such sending is accepted at most once. The query in Figure 3.1 asks this of the ciphertext `e1` that Bob receives from Alice.

> `authentication? Alice → Bob: e1` is **contradicted** if and only if, in some explored execution, one of the following holds:
>
> 1. *Origin failure.* Bob *successfully uses* `e1`, and the value Bob holds for `e1` is neither an unchanged delivery of what Alice sent nor the output of a *matching run* of Alice.
> 2. *Duplicate acceptance.* One value that Alice sent in one run is successfully used by two runs of Bob, where a run is a session clone (§5.6.3) or a scenario clone (§2.7).

Both conditions depend on *successful use*. Let U be the set of slots that Bob himself computes whose value mentions `e1` at any depth. Bob successfully uses `e1` when the explored execution reaches every slot in U and at least one of them carries a primitive that has no rewrite rule, is left unchecked, or rewrote successfully on the delivered value. A later halt therefore retracts an earlier successful decryption: if a substituted value decrypts but then fails a checked `SPLIT` or `ASSERT`, Bob has not accepted it. A defeated check (§5.2.3) is not a use either. The model's checked primitives therefore determine what constitutes acceptance (§4.5).

Verifpal evaluates the query in the recipient's state. Provenance records who created and sent each value and whether the attacker changed it (§5.5); a sender other than Alice therefore includes both forgery and redirection from another protocol action. An unchanged delivery is excluded: an attacker that forwards Alice's value intact has not impersonated her.

A *matching run* is a run of Alice that represents the same actor with the same peer binding, owns the corresponding sending, and emits the delivered value once Bob's own outgoing messages are routed into it. Session clones of one principal are always candidates; scenario clones are candidates only when the scenario left that principal's bindings unchanged (§2.7). Verifpal confirms a match by re-executing the candidate run. Routing Alice's challenge to a second session of Bob and returning that session's answer is therefore not an attack: Alice ends with one matching Bob run, nothing was forged and no sending was accepted twice. Verifpal recognizes only some matching

runs, however. A run that emits the delivered value only under a routing that Verifpal does not reconstruct is reported as an origin failure (§5.6).

This query is an injective *use-origin* test. It is not Lowe's injective agreement [11], and it is not a correspondence anchored at the completion of a run. Verifpal evaluates it at the point of use, so a responder that successfully processes an attacker-supplied first message fails the query even if it never completes the run. Conversely, a value the recipient never uses cannot fail it. Detecting a duplicate acceptance requires a second run of the sender, from a second session or from another scenario; with `--sessions 1` and no `scenarios` block, only origin failures can be detected (§5.6.3). A trace labels an origin failure as a replacement and a duplicate acceptance as a replay (§5.4).

The example queries the ciphertext `e1`, not the plaintext `m1`. Bob obtains `m1` only after `AEAD_DEC` successfully rewrites the matching `AEAD_ENC` term under the same key and nonce, so the ciphertext is the value whose origin Bob directly accepts.

3.2.3 Freshness Queries

A freshness query identifies values that can recur across runs and may therefore be replayed. A value is fresh if its resolved term contains, at any depth, a constant declared with **`generates`**.

> `freshness? v` is **contradicted** if and only if, in some principal's state in an explored execution:
>
> 1. the value `v` resolves to contains no generated constant at any depth, and
> 2. that principal *successfully uses* `v` inside at least one primitive it computes, in the same sense as in the authentication definition above.

Condition 2 excludes unused stale values, which cannot affect the protocol. Freshness depends on derivation, not secrecy. Leaking a generated value reveals it but does not remove the generated constant from its derivation. In Figure 3.2, `ha = HASH(a)` is not fresh because it depends only on the static key `a`, and Bob uses it in `ASSERT`. The query for `hb = HASH(b)` passes because `b` is generated.

The test is structural and does not test recency. A replayed authenticator still contains the generated constant of the run that produced it, so a freshness query cannot reject a stale value that was once fresh. Replays are detected by authentication queries, together with phases where the compromise is delayed.

3.2.4 Unlinkability Queries

Contact-tracing systems such as DP-3T [12], voting protocols and RFID protocols may require *unlinkability*. Definitions vary across the literature [13]. Verifpal uses the following operational goal: given two observed values, the attacker should not be able to determine whether they belong to the same principal or to different principals.

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.

Verifpal tests this goal by searching for a *link witness*: a concrete equality, runnable check or reconstruction that lets the attacker associate a pair of queried values. A queried constant is *observable* if the attacker holds its resolved value and that value traveled on the wire, was leaked, was declared public, or occurs as a subterm of a wire or leaked value. Both halves are required: an identifying check needs only a key, so without the holding condition two signatures under an unbroken key would link on shape alone. It is *secret-dependent* if its resolved value contains a generated or private constant.

> `unlinkability? c1, ..., cn` (over two or more distinct constants) is **contradicted** if and only if some pair of the queried values passes both of the following gates and admits at least one of the four witnesses below:
>
> 1. each value in the pair is observable; and
> 2. neither value in the pair is attacker-tainted: it was not installed by the attacker, and it did not arrive from a run the attacker had steered.

Gate 1 requires evidence visible in the protocol. Computing a value from a leaked seed is not an observation if that value was never sent, leaked or carried inside something the attacker opened. Gate 2 prevents the attacker from creating a link by replacing two values with the same attacker-authored term. It is stricter than authorship: a value that a steered run honestly emitted is excluded as well.

For a pair that passes both gates, Verifpal recognizes four witnesses, and reports the first that applies:

- *Observed equality.* The attacker holds two equal values that depend on a secret.

• 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)
    p1 = PUBKEY(d)
    p2 = PUBKEY(d)
  ]
Bob → Alice: h1, h2, h3, h4, h5, h6, p1, p2
queries[
    unlinkability? h1, h2, h3
    unlinkability? h4, h5, h6
    unlinkability? p1, p2
  ]
```

FIGURE 3.3 Example unlinkability queries. Only the first is uncontradicted.

Equality links them directly. The secret dependency matters because a public value common to every principal, such as a hash of public constants, identifies no one.

- *Identifying check.* Both values satisfy the same checked primitive under an identifier available to the attacker. Two signatures that verify under one public key, for example, identify the same signer. The attacker can perform the check even if no principal does so in the model.
- *Common secret origin.* The attacker can reconstruct both values from one secret-dependent value it holds. Verifpal tests reconstruction while withholding the observed values themselves; receipt alone does not establish a common origin.
- *Recognized secret.* The attacker holds at least one of the values and can run a check that identifies a secret-dependent term inside it, and that term is tied to the other value by the same kind of check or as a leaf of its reconstruction. This witness links values the attacker can recognize but cannot rebuild, such as a blind signature whose blinding factor became public.

Figure 3.3 illustrates these outcomes. The first query passes because `h1`, `h2` and `h3` depend on `a`, which the attacker does not obtain. The second is contradicted because the leaked `c` is the common origin of `h4`, `h5` and `h6`. The third is contradicted because `p1` and `p2` both resolve to the observed value `PUBKEY(d)`.

Authentication may conflict with unlinkability. Two signatures verifiable under the same

public key are linkable to the same signer. By contrast, a ring signature identifies only a member of the ring, not a particular member, so it does not create this witness. This distinction comes from the primitive specifications rather than a query-specific exception.

The four witnesses describe concrete attacker procedures; they do not define observational equivalence, which Verifpal does not analyze (§5.6). A passing query means that Verifpal found none of these witnesses, not that no distinguishing strategy exists.

3.2.5 Equivalence Queries

An equivalence query checks whether independently derived values agree in every explored execution. A common example compares the two sides of a Diffie–Hellman exchange: Alice computes `DH_KEX(gb, a)`, while Bob computes `DH_KEX(ga, b)`. An active attacker may cause these terms to diverge by replacing public keys in transit.

> `equivalence? c1, ..., cn` (over two or more distinct constants) is **contradicted** if and only if, in some explored execution, a principal's state holds every queried constant, none of them resolves to a checked primitive whose check failed, and the values they resolve to — taken after any attacker mutation and after rewriting — are not all equivalent under the equational theory of §4.8, Diffie–Hellman commutativity included.

The comparison uses the final values after attacker mutation and rewriting, not the values intended by the model author. It measures divergence in principal state rather than attacker knowledge. The relevant state must contain every queried value. If a failed check stops a principal, later values were never computed and are excluded. The failed check's output is also excluded: an unsuccessful decryption does not produce a plaintext. Stopping one principal is therefore not reported as an equivalence failure, and neither is a comparison between a value one party computed and a value the other party never reached in the same execution. Chapter 7 shows a case in which this rule matters.

A passing verdict means that the constants resolved to equivalent terms in every execution Verifpal explored. Section 2.1 includes an example.

3.2.6 Advanced Security Goals

Phases and carefully chosen queries can express compound goals such as *key-compromise impersonation resistance*, forward secrecy and post-compromise security.

Compromising Alice's long-term identity key normally lets an attacker impersonate Alice. A *key-compromise impersonation* vulnerability has a different direction: compromising Bob's long-term key lets the attacker impersonate Alice *to Bob*. Chapter 6 models this property for Signal.

Ephemeral keys can support *forward secrecy* and *post-compromise security* [7]. Forward secrecy protects messages sent before a later device compromise. Post-compromise security restores protection for messages sent after the protocol has

incorporated new uncompromised material. Both can be modeled with phases (§2.6). Chapters 6 and 8 give examples.

3.2.7 Query Options

The `precondition` option restricts a query to the executions in which a named message is sent. Written `precondition[P → T: d]` inside the brackets that follow a query, it names one message that the model contains. The query is then contradicted only by an execution in which two conditions hold at once: the query's own contradiction condition defined above, and the send of `d` from `P` to `T`, meaning that `P` reaches that send rather than halting at an earlier failed check. Both must hold in the same execution. The attacker cannot pair an honest run in which the message is sent with a mutated run in which the property fails.

The event is the send itself, a fact about the sender's own run. Whether `d` arrives at `T`, and whether `T` can be fooled about who sent it, are separate properties with their own queries. The option may be attached to any of the five query types, and a query may carry several. Adding a precondition can only remove contradictions, so a query with one is never stronger than the same query without it.

Read `Q[precondition[P → T: d]]` as: the property `Q` is a precondition for `P` sending `d` to `T`. This is the usual way to state secrecy or agreement *on acceptance*. In the following handshake, the client derives the session key before verifying the server's signature, as TLS and Noise do, and encrypts its request only after the check:

• SECRECY OF AN ACCEPTED KEY

```
attacker[active]
principal Server[
    knows private server_sk
    server_pk = PUBKEY(server_sk)
  ]
Server → Client: [server_pk]
principal Client[
    generates c
    gc = PUBKEY(c)
  ]
Client → Server: gc
principal Server[
    generates s
    gs = PUBKEY(s)
    sig = SIGN(server_sk, gs)
    k_server = DH_KEX(gc, s)
  ]
Server → Client: gs, sig
principal Client[
    k_client = DH_KEX(gs, c)
    _ = SIGNVERIF(server_pk, gs, sig)?
    generates request, n_req
    req = AEAD_ENC(k_client, n_req, request, nil)
  ]
Client → Server: n_req, req
principal Server[
    _ = AEAD_DEC(k_server, n_req, req, nil)?
  ]
queries[
    confidentiality? k_client
    confidentiality? k_client[
        precondition[Client → Server: req]
      ]
  ]
```

The first query is contradicted. The attacker replaces `gs` with its own public key, so the client computes a key the attacker knows, and then halts at the signature check without using it. This is a true statement about a value the client discarded. The second query holds: in every execution in which the client goes on to send `req`, the signature verified, so `gs` was the server's and the key is a two-party secret. Without the option, the only way to ask this is to query a value derived after the check, such as `request`, and to argue separately that the answer carries back to the key.

```
equivalence? k_client, k_server[precondit
```

The same shape states key agreement on completion, as in [11] and it conditions an authentication query on the recipient's onward action. In the following model, the intended property is that Alice sends `m2` to Carol only after authenticating Bob's ciphertext:

• 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)
  ]
```

Add the option to the authentication query for `e`:

• QUERY OPTIONS EXAMPLE (CONT.)

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

The query is contradicted if Alice accepts an unauthenticated `e` and still sends `m2` to Carol. In this model, the attack that Verifpal reports is a duplicate acceptance: the attacker replays `e` and `h` from Bob's other session, Alice's MAC check passes because the pre-shared key is the same in both sessions, and Alice forwards the replayed plaintext. The trace ends with a line stating that Alice still sends `m2` to Carol, so the failure counts. Had Alice halted before sending `m2`, the execution would fall outside the precondition and contradict nothing.

Here, `m2` is guarded in transit to Carol, so a query on that message alone would not reveal the earlier failure. The precondition relates the guarded onward action to the authentication of `e`.

`precondition` is the only query option in this version of Verifpal.

3.3 The Attacker

Verifpal models a network attacker, derives everything available to it under the symbolic equations and tests whether any explored execution contradicts a query.

A *passive attacker* observes all network traffic. It repeatedly decomposes known terms, constructs terms from known components and identifies equivalent terms until deduction reaches a fixed point. For example, it decrypts a ciphertext when it knows the required key. It cannot change a message, so Verifpal analyzes one honest execution per phase.

An *active attacker* may also replace any unguarded value in transit with an observed or constructed term. Each set of replacements defines a mutated execution. Instead of enumerating all sets, Verifpal works backward from unresolved queries to derive the substitutions needed for a contradiction (Chapter 5). It then executes each candidate concretely, computes the deduction fixed point and evaluates the queries.

Use the passive model only when the deployment prevents traffic modification or when the analysis deliberately studies observation alone. Most network protocols, including HTTPS, secure messaging, VPNs and SSH, require the active attacker model.

An active attacker has the following capabilities.

Modify messages. The attacker may replace `e1` with `e2`, or another constructible value, while it travels from Alice to Bob. Bob receives the replacement, while Alice's local state retains the original `e1`. The attacker cannot alter local state directly. It can read but not replace guarded constants (§2.5), except when the forwarding sender previously received that constant without a guard.

Construct and inject values. The attacker may replace Alice's `PUBKEY(a)` with an attacker-controlled public key whose private value it knows. Section 2.1 shows why unauthenticated key exchange is vulnerable to this substitution. The attacker builds every injected term from constants that already exist in the model, such as `nil` and the values it has observed; it cannot invent a fresh nonce of its own (§5.3).

Route traffic across sessions. By default, every principal has two concurrent sessions. Each session has distinct generated values, while values declared with **`knows`** are shared (§5.6.3). The attacker may route traffic between sessions, reuse one session to answer another's challenge and replay messages protected by shared long-term keys. Because authentication queries reject duplicate acceptance (§3.2.2), one sending accepted twice is a contradiction even without forgery. The attacker retains knowledge across executions, but every injected term must have been available to it before the receive it replaces (§5.2.3). Analysis remains bounded by the selected session count.

3.3.1 Checked Primitives and Truncation

Checked primitives (§4.5) affect honest and mutated executions differently.

Under a passive attacker, a failed check occurs in the honest execution. It therefore indicates an inconsistent model, and Verifpal reports an error.

Under an active attacker, a check may fail only because of substituted network values. Verifpal *truncates* that principal's execution at the failure and omits its later operations. Earlier operations and all knowledge obtained from them remain part of the analysis.

During backward search, Verifpal therefore seeks substitutions that satisfy the checks needed to reach a query contradiction. Chapter 5 describes this process.

Guards and checked primitives express different controls. A guard models prior authentication of a network value, such as a long-term public key. A checked `SIGNVERIF` or `AEAD_DEC` models a protocol that stops when verification or authenticated decryption fails. Chapter 6 analyzes the effect of each control.

Queries Require Successful Use

An authentication query concerns successful use by the recipient. If Bob never reads or uses Alice's message, Verifpal cannot find an execution in which Bob accepts an unauthenticated copy. A passing verdict in this case does not show that the unused message was authenticated. It shows only that no explored execution met the contradiction conditions.

3.4 Reading Results

Figure 3.4 combines these concepts in a challenge–response protocol. The client generates a fresh nonce[1] and asks the server to sign it with `s`, whose public key is `gs`. A valid signature under the authenticated public key demonstrates access to `s`. The client then signs an attestation, which the server verifies.

Verifpal reports the following results:

[1] A *nonce* is a value intended for one use.

CHALLENGE-RESPONSE PROTOCOL

```
attacker[active]
principal Server [
    knows private s
    gs = PUBKEY(s)
  ]
principal Client[
    knows private c
    gc = PUBKEY(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 3.4 A simple challenge-response protocol in Verifpal.

• CHALLENGE-RESPONSE: INITIAL RESULTS

```
Fail × authentication? Server → Client: proof
Attack trace:
| 1. Attacker observes nonce on the wire.
| 2. Attacker constructs SIGN(nil, nonce).
| 3. Attacker replaces proof (sent by Server to Client) with
| SIGN(nil, nonce). (proof was SIGN(s, nonce))
> proof (SIGN(nil, nonce)), sent by Attacker and not by Server,
is successfully used in SIGNVERIF(gs, nonce, proof) within
Client's state.
Fail × authentication? Client → Server: signed
Attack trace:
| 1. Attacker observes attestation#2 on the wire.
| 2. Attacker observes signed#2 on the wire, where it is
| SIGN(c, attestation#2).
| 3. Attacker replays attestation (Client to Server) from
| another session, where it is attestation#2.
| 4. Attacker replays signed (Client to Server) from another
| session, where it is SIGN(c, attestation#2).
| 5. Server's SIGNVERIF(gc, attestation#2, SIGN(c, attestation#2))?
| passes — the attacker controls one of its inputs.
| 6. Attacker knows nil: it is public.
> signed (SIGN(c, attestation#2)), which Client sent in another
session and not in this one, is successfully used in
SIGNVERIF(gc, attestation, signed)? within Server's state:
Client sent it once, Server accepts it twice, so agreement
is not injective.

Fail × 2 of 2 queries failed.
```

The first trace is an origin failure. Its conclusion identifies the violated condition: Client successfully used the attacker's `proof`, rather than Server's, in `SIGNVERIF`. Steps 1–3 explain the construction. The attacker observes `nonce`, signs it with the known private value `nil` and substitutes the resulting signature for `proof`. The parenthetical term records the honest value that was replaced. The forged signature does not verify under `gs`, but Client does not check `SIGNVERIF`: it computes `valid` and continues regardless of the outcome, so any value counts as used.

Adding `?` to the verification alone does not repair the protocol. Client now stops on an invalid signature, and the attacker instead replaces the key together with the signature:

• CHALLENGE-RESPONSE: CHECKED, BUT UNGUARDED

```
Fail × authentication? Server → Client: proof
Attack trace:
| 1. Attacker constructs PUBKEY(nil).
| 2. Attacker observes nonce on the wire.
| 3. Attacker constructs SIGN(nil, nonce).
| 4. Attacker replaces gs (sent by Server to Client) with
| PUBKEY(nil). (gs was PUBKEY(s))
| 5. Attacker replaces proof (sent by Server to Client) with
| SIGN(nil, nonce). (proof was SIGN(s, nonce))
| 6. Client's SIGNVERIF(PUBKEY(nil), nonce, SIGN(nil, nonce))?
| passes — the attacker controls one of its inputs.
> proof (SIGN(nil, nonce)), sent by Attacker and not by Server,
is successfully used in SIGNVERIF(gs, nonce, proof)? within
Client's state.
```

Step 6 is a *gate*: a checked primitive that passed on attacker-controlled input. Because `gs` is unguarded in the same message, the attacker substitutes its own public key and a signature under the matching private value. Backward search derives the key substitution from the requirement that the check succeed; it does not enumerate pairs of replacements (§5.2.3). Guarding `gs` models Client having authenticated Server's signing key before this exchange. Guarding the key without checking the signature also fails, because an unchecked Client accepts `nil` in place of `proof`.

After guarding `gs` and checking `SIGNVERIF`, the first query passes and the second still fails:

• CHALLENGE-RESPONSE: AFTER BOTH FIXES

```
Pass ✓ authentication? Server → Client: proof  [search exhausted at 2 sessions]
Fail × authentication? Client → Server: signed
Attack trace:
| 1. Attacker observes attestation#2 on the wire.
| 2. Attacker observes signed#2 on the wire, where it is
| SIGN(c, attestation#2).
| 3. Attacker replays attestation (Client to Server) from
| another session, where it is attestation#2.
| 4. Attacker replays signed (Client to Server) from another
| session, where it is SIGN(c, attestation#2).
| 5. Server's SIGNVERIF(gc, attestation#2, SIGN(c, attestation#2))?
| passes — the attacker controls one of its inputs.
| 6. Attacker knows nil: it is public.
> signed (SIGN(c, attestation#2)), which Client sent in another
session and not in this one, is successfully used in
SIGNVERIF(gc, attestation, signed)? within Server's state:
Client sent it once, Server accepts it twice, so agreement
is not injective.

Fail × 1 of 2 queries failed.
```

The remaining trace is a duplicate acceptance. The attacker forges nothing. It records the attestation and signature that Client sent in its second session, whose values carry the `#2` suffix, and delivers them to the first session of Server. The signature verifies because `gc` is long-term and shared by both sessions. Client sent this attestation once, and two Server sessions accept it. The protocol as modeled does not bind the attestation to the server's session: nothing that Client signs depends on a value chosen by Server, so any Server session accepts a recorded attestation.

A single-session analysis cannot represent this execution because it has no second Client session to record from:

• CHALLENGE-RESPONSE: AFTER BOTH FIXES, AT ONE SESSION

```
Pass ✓ authentication? Server → Client: proof  [search exhausted at 1 session]
Pass ✓ authentication? Client → Server: signed  [search exhausted at 1 session]

Pass ✓ All 2 queries pass.
```

The two verdicts for the second query do not conflict. Each is relative to its envelope, and the one-session envelope excludes every attack that needs two runs of a role (§5.6.3). Compare a passing verdict with its envelope before drawing a conclusion from it.

The corrected model states both trust assumptions explicitly and exposes a replay that the original queries were meant to exclude. A productive modeling cycle is to write the protocol and queries, inspect every trace, revise either the design or an inaccurate assumption, and rerun the analysis. Part III applies this process to larger protocols.

PART II

The Verifpal Reference

CHAPTER 4

Primitives and the Equational Theory

This chapter defines Verifpal's built-in primitives, their attacker deduction rules and their equations. Section 4.2 explains the common specification format. The catalog documents each primitive, and Table 4.1 provides a compact reference.

4.1 Primitives in the Symbolic Model

Verifpal represents cryptographic primitives as ideal operations in the Dolev–Yao symbolic model [4]. A hash is an uninterpreted function: the attacker obtains `HASH(x)` by knowing `x` and applying `HASH`. The model does not represent collisions or length-extension attacks. Likewise, the plaintext in `ENC(k, m)` can be recovered only with `k`; key length and cipher mode are outside the model.[1]

4.2 Primitive Specifications

Verifpal defines each primitive with a common PrimitiveSpec. A specification records the name, input arity, output count and a combination of six rule types:

- Decompose. Given a held primitive term and specified inputs, reveal one or more other inputs or sibling outputs. For example, `ENC(k, m)` and `k` reveal `m`, and the ciphertext output of `KEM_ENCAP` together with the decapsulation key reveals both the shared secret and the encapsulation randomness. A rule reveals only the specified values, and only from the held term: knowing `k` does not yield `m` from a ciphertext the attacker never obtained. Without both `k` and the nonce `n`, the opaque term `AEAD_ENC(k, n, m, ad)` reveals neither `m` nor `ad`.
- Recompose. Given enough distinct outputs of a primitive, recover an input. Any three distinct outputs of `a, b, c, d, e = THRESHOLD_SPLIT[3](x)`, for example, reveal `x` during attacker deduction; the number is the threshold the split carries in its bracket.
- Rewrite. Reduce a primitive expression whose arguments match a pattern. The rule `DEC(k, ENC(k, m))` → `m` represents successful decryption. Rewrite rules define algebraic relations between operations.

[1] Verifpal, ProVerif and Tamarin use symbolic models. Computational tools such as CryptoVerif [5] reason about concrete cryptographic assumptions and parameters.

- Rebuild. Reduce a primitive whose inputs are outputs from the same instance of another primitive. If `a` and `b` are distinct outputs of `THRESHOLD_SPLIT[2](x)`, then `THRESHOLD_JOIN(a, b)` reduces to `x`. Rebuilding occurs while resolving principal state; recomposition occurs during attacker deduction.
- Combine. Reduce a primitive whose inputs each wrap a distinct output of one split, agreeing on stated positions, into the same wrapper applied to the split's secret. Enough `THRESHOLD_SIGN` partials over one message combine through `THRESHOLD_JOIN` into `SIGN(k, message)`, and enough `PUBKEY(share)` values into `PUBKEY(k)`. The rule runs in both directions: a principal's join reduces by it, and the attacker obtains the whole from partials it holds or can produce (§4.3.7).
- Reuse. Name a set of argument positions that must stay unique across calls, and state what two calls agreeing on all of them give away. `AEAD_ENC` declares this rule over its key and nonce: two ciphertexts sharing them surrender both plaintexts and permit a forgery under that key and nonce (§4.4). `THRESHOLD_SIGN` declares it over its share and nonce: two partials sharing them surrender the share (§4.3.7).

A specification may also forbid particular argument forms (§4.7), declare commutativity and define the effect of weakening assumptions (§4.6). These declarations fully describe Diffie–Hellman without special language syntax. The analysis engine reads primitive behavior from the specifications. Only the reductions for the core operations `ASSERT`, `CONCAT` and `SPLIT` are implemented directly.

4.3 The Catalog

4.3.1 Core Primitives

Core primitives perform structural operations rather than cryptographic ones:

- `ASSERT(MAC(key, message), MAC(key, message)): unused`.
  Compares two values for equality. It is commonly used to compare MAC values. Its output is not meaningful; add `?` to enforce the comparison (§4.5).
- `CONCAT(a, b...): c`.
  Concatenates between two and five values into one value.
- `SPLIT(CONCAT(a, b)): a, b`.
  Splits a concatenation into its components. Its input must be a `CONCAT` term.

4.3.2 Hashing Primitives

Hashing primitives represent unkeyed hashing, message authentication and key derivation:

- `HASH(a, b...): x`.
  A secure hash function, comparable to BLAKE2s [14]. It accepts one to five inputs and produces one output.

- `MAC(key, message): hash`.
  A keyed message-authentication function.
- `HKDF(salt, ikm, info): a, b...`.
  A hash-based key-derivation function modeled on HKDF [15]. It derives one to five outputs from input key material, with salt and context information.

4.3.3 Public-Key Primitives

Two primitives represent public keys and Diffie–Hellman key exchange. Section 4.7 gives further details.

- `PUBKEY(key): pubkey`.
  Derives a public key from a private value. In a Diffie–Hellman setting, `PUBKEY(a)` denotes g^a. The same constructor is used for signatures, public-key encryption, ring signatures and key encapsulation. A public key cannot be passed to `PUBKEY`.
- `DH_KEX(PUBKEY(a), b): shared secret`.
  Derives the Diffie–Hellman shared secret g^{ab} from a peer's public key and a local private value. Verifpal equates `DH_KEX(PUBKEY(a), b)` with `DH_KEX(PUBKEY(b), a)`. The second argument cannot be a public key, and `DH_KEX` cannot be nested inside itself.

4.3.4 Encryption Primitives

Encryption primitives cover symmetric encryption, authenticated encryption and public-key encryption:

- `ENC(key, plaintext): ciphertext`.
  Symmetric encryption without authentication, comparable to AES-CBC or ChaCha20. Because it provides no ciphertext integrity, `ENC` accepts the **`malleable`** assumption defined in §4.6.
- `DEC(key, ENC(key, plaintext)): plaintext`.
  Symmetric decryption.
- `AEAD_ENC(key, nonce, plaintext, ad): ciphertext`.
  Authenticated encryption with associated data, comparable to AES-GCM or ChaCha20-Poly1305. Decryption succeeds only with the same key, nonce and associated data. The nonce must be unique for each encryption under a key; §4.4 defines what Verifpal derives from a repeated one. Verifpal treats the ciphertext as opaque and does not reveal `ad`; send the associated data separately if the attacker should observe it.
- `AEAD_DEC(key, nonce, ciphertext, ad): plaintext`,
  where the ciphertext is `AEAD_ENC(key, nonce, plaintext, ad)`.
  Authenticated decryption with associated data. The nonce is an input to decryption, so a leaked key alone opens nothing: the attacker needs the nonce as well. Add `?` to stop the principal when authentication fails (§4.5).

- `PKE_ENC(PUBKEY(key), plaintext): ciphertext`.
  Public-key encryption. Opening it requires the private `key`, not the public value it was encrypted to.
- `PKE_DEC(key, PKE_ENC(PUBKEY(key), plaintext)): plaintext`.
  Public-key decryption.

4.3.5 Signature Primitives

Signature primitives cover ordinary, ring and blind signatures:

- `SIGN(key, message): signature`.
  Signs a message with a private key. The corresponding public key is `PUBKEY(key)`.
- `SIGNVERIF(PUBKEY(key), message, SIGN(key, message)): verified`.
  Verifies a signature against a public key and message. Its successful output is a verification token, not the signed message. Add `?` to stop the principal if verification fails (§4.5).
- `RINGSIGN(key_a, PUBKEY(key_b), PUBKEY(key_c), message): signature`.
  Creates a three-member ring signature [16]. The first argument is the actual signer's private key; the next two are the other members' public keys. Verification proves that one ring member signed without identifying that member.
- `RINGSIGNVERIF(PUBKEY(a), PUBKEY(b), PUBKEY(c), m, s): verified`,
  where `s = RINGSIGN(a, PUBKEY(b), PUBKEY(c), m)`.
  Verifies a ring signature with all three public keys. Their order need not match the signing call, but the three verifier keys must correspond one-to-one to the ring that was signed; a ring collapsed onto a repeated key does not verify. Add `?` to enforce successful verification.
- `BLIND(k, m): blinded`.
  Blinds `m` with the secret factor `k`, allowing a signer to sign the blinded term without learning `m`.
- `UNBLIND(k, m, SIGN(a, BLIND(k, m))): SIGN(a, m)`.
  Converts a signature over the blinded message into a signature over `m`. The signature is the third argument. Unblinding needs no secret of the signer: anyone holding `k`, `m` and the blind signature obtains `SIGN(a, m)`, so the blinding factor must stay secret.

4.3.6 Key-Encapsulation Primitives

The generic key-encapsulation mechanism can model post-quantum schemes such as ML-KEM [17]:

- `KEM_ENCAP(ek, r): ss, ct`.
  Given `ek = PUBKEY(dk)` and fresh randomness `r`, produces a shared secret `ss` and encapsulation ciphertext `ct`. Randomness is an explicit argument. Holding `ct` and `dk` reveals both `ss` and `r`, because a decapsulator that performs the re-encryption check of ML-KEM re-derives the randomness. Holding `ss` alone reveals nothing.

- `KEM_DECAP(dk, ct): ss`.
  Recovers the shared secret from a matching ciphertext and private decapsulation key. The first argument cannot be a public key, and the second must be the ciphertext output rather than the shared secret.

Encapsulation Randomness Is Required

Every primitive is a deterministic symbolic function of its arguments. Without explicit randomness, two encapsulations to the same key would produce the same symbolic secret and ciphertext, unlike a real randomized KEM.
Declare `r` with **`generates`** and use a distinct generated value for each encapsulation. Reuse models repeated encapsulation randomness and may invalidate the analysis.

4.3.7 Threshold Primitives

Verifpal models *t*-of-*n* Shamir secret sharing [18] and the threshold Schnorr signatures built on it, in the shape of FROST [19]:

- `THRESHOLD_SPLIT[t](k): s1, ..., sn`.
  Shares `k` so that any `t` of the `n` bound shares recover it and fewer reveal nothing. The bracket carries the threshold, which is required and must lie between 2 and `n`; `n` is the number of constants bound on the left, at most 16. The threshold is part of the term: shares of a `THRESHOLD_SPLIT[2](k)` and of a `THRESHOLD_SPLIT[3](k)` are different values and never combine.
- `THRESHOLD_JOIN(x1, ..., xm): k`.
  Lagrange interpolation over pieces of one split. Given at least `t` distinct shares it yields the secret; given at least `t` partial signatures over distinct shares, the same commitments and the same message, it yields `SIGN(k, message)`; given at least `t` values `PUBKEY(si)` it yields `PUBKEY(k)`. Fewer than `t` pieces, a repeated share, or partials that disagree leave the term unreduced, which a later `SIGNVERIF` rejects.
- `THRESHOLD_SIGN(share, nonce, cl, message): partial`.
  One signer's share of a threshold signature, with the inputs of FROST's second signing round other than the participant identifier, which the share's position carries, and the group public key, which the share determines: the signer's share, its own fresh nonce, the commitment list `cl` the coordinator distributed, and the message. The commitment list is any term the model builds, typically a `CONCAT` of the participants' `PUBKEY(nonce)` commitments; partials from different signing sessions carry different lists and do not combine, which is what FROST's binding factor, a hash over the group key, the message and the commitment list, enforces. FROST itself must not derive its nonces deterministically; a model of some other scheme whose partials are deterministic simply generates a fresh nonce per call.

The joined signature is a plain `SIGN(k, message)`, verified by `SIGNVERIF` under `PUBKEY(k)` exactly as a FROST signature verifies under ordinary Schnorr verification, and it does not reveal which shares produced it. The attacker recovers a secret from `t` shares it holds, builds a signature from `t` partials it holds or can produce, including partials it obtains by sending a message of its own to a participant that signs whatever it receives, and derives the group key from verification shares. RFC 9591 forbids using a nonce as input to more than one signing and warns that a replay permits a complete key-recovery attack (its Section 7.3). Verifpal's rule is the plain-Schnorr reading of that warning, with one nonce standing for FROST's hiding-and-binding pair: two partials under one share and one nonce, whatever else differs, give the share to whoever holds both, and `t` shares so recovered are the key. A participant therefore declares its nonce with **`generates`** for every signing.

Key generation is a dealer's split. A distributed key generation is modelled by a principal that splits a **`generates`** key and never uses it otherwise. Figure 4.1 is FROST as RFC 9591 lays it out, at a two-of-three threshold: the RFC's dealer (its Appendix C) sends each share over a mutually authenticated confidential channel and deletes the key afterwards, which the model stands in for with a pre-shared key, and the model additionally binds each share to the group key, a choice of the model rather than of the RFC, so that a share cannot be replayed into a later key generation; each participant commits to a fresh nonce; the coordinator picks the message and distributes each participant's commitment to the others over authenticated channels; each participant assembles the commitment list around its own commitment, which is how the RFC's requirement that a signer find its own commitment in the list (its Section 5.2) is met, and returns a partial; and the coordinator interpolates them into a signature the verifier checks under `PUBKEY(k)`. All three queries hold. Remove the guards on the coordinator's second-round messages and the network is a rogue coordinator: it relays the honest commitments, collects partials over a message of its own, and the verifier accepts a signature the coordinator never produced, which is the RFC's demand for authenticated channels turned into a finding. Leak a third share beside one unguarded participant and the same forgery needs no second oracle.

4.4 Nonces and Nonce Reuse

A nonce is an ordinary Verifpal term. Declare it with **`generates`** for a value that is fresh in every session, or with **`knows private`** for one that every session of the declaring principals shares (§5.6.3). Because the nonce is an input to decryption, the recipient must have it, so a model normally sends it beside the ciphertext or derives it on both sides from shared material.

Real authenticated encryption requires that a nonce be used once per key. Verifpal does not enforce that requirement; it analyzes the consequence of breaking it. Two `AEAD_ENC` terms form a *reused pair* when they agree on key and nonce and are not the same term. Two encryptions of one plaintext under the same key, nonce and associated data are one term written twice, so they are not a pair. Two encryptions that differ only in their associated data are.

• EXAMPLE: FROST AT A TWO-OF-THREE THRESHOLD

```
attacker[active]
principal Dealer[
    knows private psk_a, psk_b
    generates k, nd_a, nd_b
    pk = PUBKEY(k)
    s1, s2, s3 = THRESHOLD_SPLIT[2](k)
    ea = AEAD_ENC(psk_a, nd_a, s1, pk)
    eb = AEAD_ENC(psk_b, nd_b, s2, pk)
  ]
Dealer → Alice: [pk], nd_a, ea
Dealer → Bob: [pk], nd_b, eb
Dealer → Coordinator: [pk]
Dealer → Verifier: [pk]
principal Alice[
    knows private psk_a
    sa = AEAD_DEC(psk_a, nd_a, ea, pk)?
    generates na
    ca = PUBKEY(na)
  ]
principal Bob[
    knows private psk_b
    sb = AEAD_DEC(psk_b, nd_b, eb, pk)?
    generates nb
    cb = PUBKEY(nb)
  ]
Alice → Coordinator: ca
Bob → Coordinator: cb
principal Coordinator[
    generates m
  ]
Coordinator → Alice: [m], [cb]
Coordinator → Bob: [m], [ca]
principal Alice[
    cl_a = CONCAT(ca, cb)
    pa = THRESHOLD_SIGN(sa, na, cl_a, m)
  ]
principal Bob[
    cl_b = CONCAT(ca, cb)
    pb = THRESHOLD_SIGN(sb, nb, cl_b, m)
  ]
Alice → Coordinator: pa
Bob → Coordinator: pb
principal Coordinator[
    sig = THRESHOLD_JOIN(pa, pb)
    _ = SIGNVERIF(pk, m, sig)?
  ]
Coordinator → Verifier: m, sig
principal Verifier[
    _ = SIGNVERIF(pk, m, sig)?
  ]
queries[
    confidentiality? k
    confidentiality? s1
    authentication? Coordinator → Verifier: sig
  ]
```

FIGURE 4.1 FROST's two signing rounds at a two-of-three threshold. The joined signature is a plain `SIGN(k, m)`, and the verifier needs nothing but the message and the group key.

• EXAMPLE: A REPEATED NONCE

```
attacker[passive]
principal Alice[
    knows private k
    knows private n1, n2
    knows private m1, m2, m3
    knows public ad
    e1 = AEAD_ENC(k, n1, m1, ad)
    e2 = AEAD_ENC(k, n1, m2, ad)
    e3 = AEAD_ENC(k, n2, m3, ad)
  ]
Alice → Bob: e1, e2, e3
principal Bob[
    knows private k
    knows private n1, n2
    knows public ad
    d1 = AEAD_DEC(k, n1, e1, ad)?
    d2 = AEAD_DEC(k, n1, e2, ad)?
    d3 = AEAD_DEC(k, n2, e3, ad)?
  ]
queries[
    confidentiality? m1
    confidentiality? m2
    confidentiality? m3
  ]
```

FIGURE 4.2 Alice encrypts the first two messages under one nonce and the third under another.

A reused pair has two consequences, matching what AES-GCM and ChaCha20-Poly1305 lose when a nonce repeats:

- *Plaintext recovery.* The attacker obtains the plaintext of both ciphertexts in the pair.
- *Forgery.* The attacker constructs `AEAD_ENC(key, nonce, m, ad)` for any `m` and `ad` it can obtain, without holding the key or the nonce.

Both are scoped to the pair. The key is not recovered, and a message under the same key with a different nonce is unaffected:

• A REPEATED NONCE

```
Fail × confidentiality? m1
Attack trace:
| 1. Attacker observes e1 on the wire.
| 2. Attacker observes e2 on the wire.
| 3. Attacker recovers m1 from e1: e2 shares its key and
| nonce.
> m1 (m1) is obtained by Attacker.
Fail × confidentiality? m2
Attack trace:
| 1. Attacker observes e2 on the wire.
| 2. Attacker observes e1 on the wire.
| 3. Attacker recovers m2 from e2: e1 shares its key and
| nonce.
> m2 (m2) is obtained by Attacker.
Pass ✓ confidentiality? m3  [search exhausted at 2 sessions]

Fail × 2 of 3 queries failed.
```

The attacker here is passive: it alters nothing and only records the two ciphertexts. `m3` stays confidential because `n2` is used once.

Recovery is an over-approximation in one case. Two plaintexts encrypted under one keystream give a real attacker their combination, from which each plaintext follows only when the other is known or guessable. The symbolic model has no such combination, so Verifpal hands the attacker both plaintexts outright. A model whose two reused plaintexts are both unknown therefore reports a disclosure that a concrete attacker would still have work to do to complete. This is a deliberate modeling choice: the analysis errs toward reporting the loss.

The forgery is the authenticity half. In the following model every ciphertext uses one nonce, and the associated data of the attacked message differs from that of the other two, so no honest ciphertext can be substituted in its place:

• FORGING UNDER A REPEATED NONCE

```
Fail × authentication? Alice → Bob: e3
Attack trace:
| 1. Attacker observes e1 on the wire.
| 2. Attacker observes e2 on the wire.
| 3. Attacker forges AEAD_ENC(k, n, nil, ad2) under the key
| and nonce shared by e1 and e2, using nil, ad2.
| 4. Attacker replaces e3 (sent by Alice to Bob) with
| AEAD_ENC(k, n, nil, ad2). (e3 was AEAD_ENC(k, n, m3, ad2))
| 5. Bob's AEAD_DEC(k, n, AEAD_ENC(k, n, nil, ad2), ad2)?
| passes — the attacker controls one of its inputs.
> e3 (AEAD_ENC(k, n, nil, ad2)), sent by Attacker and not by
Alice, is successfully used in AEAD_DEC(k, n, e3, ad2)?
within Bob's state.
```

• EXAMPLE: FORGING UNDER A REPEATED NONCE

```
attacker[active]
principal Alice[
    knows private k
    knows private n
    knows public ad1, ad2
    generates m1, m2, m3
    e1 = AEAD_ENC(k, n, m1, ad1)
    e2 = AEAD_ENC(k, n, m2, ad1)
    e3 = AEAD_ENC(k, n, m3, ad2)
  ]
Alice → Bob: e1, e2, e3
principal Bob[
    knows private k
    knows private n
    knows public ad1, ad2
    d = AEAD_DEC(k, n, e3, ad2)?
    r = HASH(d)
  ]
queries[
    authentication? Alice → Bob: e3
  ]
```

FIGURE 4.3 The attacker holds no key, yet Bob accepts a ciphertext Alice never sent.

Step 3 is the reuse rule. The attacker supplies the plaintext and the associated data, both public here, while the key and nonce positions are exempt because `e1` and `e2` share them. Bob's checked decryption then accepts a ciphertext produced without any secret at all.

The two ciphertexts of a pair must be available in one execution. Verifpal confirms this before pairing them, so two values of one slot observed in two different mutated executions do not form a reused pair.

One Nonce, One Message

Give every encryption under a key its own **`generates`** nonce unless the model means to hold the ciphertext fixed. A nonce declared with **`knows`** is shared by every session of the principals that declare it, so two sessions encrypting different plaintexts under it are a reused pair, and the analysis will say so.

This mirrors deployment. A counter, a random nonce and a nonce derived from a key schedule are each modeled by choosing which declaration the nonce gets.

4.5 Checked Primitives

Six primitives are *checkable*: `ASSERT`, `SPLIT`, `AEAD_DEC`, `SIGNVERIF`, `RINGSIGNVERIF` and `KEM_DECAP`. A question mark after a call makes it *checked*. If its rewrite rule does not match, the principal stops at that operation. This models rejection after unequal assertions, failed authenticated decryption or invalid signatures. A checked `KEM_DECAP` stops the principal when the ciphertext does not match its key; an unchecked call continues with the unreduced decapsulation term. The symbolic KEM is deterministic and does not model the implicit rejection of ML-KEM or any probabilistic behavior.

For example, `SIGNVERIF(k, m, s)?` requires successful verification. A rewrite rule does not by itself make a primitive checkable. `DEC`, `PKE_DEC` and `UNBLIND` rewrite, but cannot be checked because a mismatched input produces a different symbolic value rather than a detectable failure. Section 3.3 explains how checks affect attacker search.

When to Check Primitives

A checked primitive asserts that the implementation detects failure and stops. Add `?` only when the actual protocol makes that decision. Leaving a call unchecked models an implementation that continues with the result. Comparing both models can show the security effect of a required validation step.

4.6 Declared Weakening Assumptions

By default, every primitive satisfies its ideal symbolic rules: opening `ENC(k, m)` requires `k`, and producing `SIGN(sk, m)` requires `sk`. A *declared weakening assumption* lets a model analyze the loss of a specific guarantee, either immediately or in a later phase.

Place the assumption in brackets between the primitive name and its arguments:

• EXAMPLE: WEAKENING ASSUMPTIONS

```
principal Alice[
    knows private sk, k, m
    knows public ad
    generates a, n
    ga = PUBKEY[weak](a)
    s = SIGN[forgeable](sk, m)
    c = ENC[malleable](k, m)
    e = AEAD_ENC[weak from phase 2](k, n, m, ad)
  ]
```

Each entry grants one defined attacker capability. An unannotated model retains the ideal rules. Section 4.6.4 explains when an annotation also affects equivalent terms or other calls under the same secret.

4.6.1 The Three Capabilities

Three keywords represent distinct failures:

- `weak` removes confidentiality. Holding the annotated term reveals its protected input or output. For `HASH`, it reveals every preimage argument. For `AEAD_ENC`, `ENC` and `PKE_ENC`, it reveals the plaintext. For `KEM_ENCAP`, it reveals the shared secret. For `PUBKEY`, it reveals the private key.
- `forgeable` removes authenticity. The attacker can construct the primitive without its secret argument. This assumption is available for `SIGN`, `MAC`, `RINGSIGN` and `AEAD_ENC`. The secret argument of `AEAD_ENC` is its key; the nonce, plaintext and associated data must still be obtainable, which is what separates this assumption from the forgery a repeated nonce grants (§4.4).
- `malleable` removes ciphertext integrity. Given `ENC(k, m)`, the attacker may construct `ENC(k, m2)` for any constructible `m2` without learning `k`. Only `ENC` supports this assumption.

Separating confidentiality from authenticity allows `AEAD_ENC[forgeable]` to model an attacker that can create an accepted ciphertext without being able to read an honest ciphertext.

The `malleable` assumption modifies an existing ciphertext; it does not create a ciphertext under an unknown key from nothing. The attacker must first hold a matching ciphertext and may vary only the designated plaintext position. The assumption applies to the annotated term and its equivalents, not to every `ENC` call. Thus, annotating `ENC(k1, m)` does not weaken `ENC(k2, m)`. This abstraction is stronger than a concrete cipher mode in one respect and weaker in another: it grants an arbitrary chosen replacement of the plaintext, but it cannot express a correlated change to an unknown plaintext, such as flipping selected bits of `m`.

`PUBKEY[weak]` models recovery of a private value from its public key, including a future discrete-logarithm break. After recovering `a` from `ga`, the attacker derives affected `DH_KEX` terms through ordinary reconstruction (§5.2.1). Chapter 8 uses this annotation.

4.6.2 Phase-Scoped Assumptions

Append `from phase N` to activate an assumption at phase `N`. It remains active in every later phase.

This qualifier can model a harvest-now, decrypt-later attacker. A ciphertext sent in phase 0 under `weak from phase 2` becomes readable in phase 2:

• EXAMPLE: HARVEST NOW, DECRYPT LATER

```
principal Alice[
    knows private k, m_now, m_later
    generates n_now, n_later
    e_now   = AEAD_ENC(k, n_now, m_now, nil)
    e_later = AEAD_ENC[weak from phase 2](k, n_later, m_later, nil)
  ]

Alice → Bob: n_now, e_now, n_later, e_later
```

• HARVEST NOW, DECRYPT LATER

```
Warning ▲ Analysis performed under 1 declared weakening assumption:
Warning ▲ AEAD_ENC[weak from phase 2](k, n_later, m_later, nil)

Pass ✓ confidentiality? m_now  [search exhausted at 2 sessions]
Fail × confidentiality? m_later
Attack trace:
| 1. Attacker observes e_later on the wire.
| 2. Attacker breaks e_later under the declared `weak'
| assumption, obtaining m_later.
> m_later (m_later) is obtained by Attacker.

Fail × 1 of 2 queries failed.
```

The `leaks` declaration cannot express every such failure. It can reveal a key in a later phase, but it cannot grant the ability to invert a hash because that ability is not a constant.

A later capability does not permit retroactive substitution. If a signature is sent in phase 0 and becomes forgeable in phase 3, the attacker cannot replace the earlier message. Phase rules still determine when a substitution may occur (§2.6). The annotation changes which terms the attacker can construct, not the time of an action.

4.6.3 What Verifpal Rejects

Verifpal rejects an assumption that the primitive does not support. When possible, the diagnostic suggests the applicable assumption:

- `SIGN[weak]` is invalid because a signature provides authenticity, not confidentiality. Use `SIGN[forgeable]` to model an authenticity failure.
- `DH_KEX[weak]` is invalid because discrete-logarithm recovery applies to a public key, not to the exchange operation. Annotate the corresponding `PUBKEY` call instead.
- `HASH[forgeable]` is invalid because `HASH` has no secret argument; anyone who knows the inputs can compute it.
- `AEAD_ENC[malleable]` is invalid because malleability of an authenticated primitive is an authenticity failure. Use `AEAD_ENC[forgeable]`. The same distinction applies to `MAC` and `SIGN`.

- `HASH[malleable]` is invalid because a hash is not a ciphertext.
- No capability applies to `ASSERT`, `CONCAT` or `SPLIT`; these core primitives provide no cryptographic guarantee to weaken.
- An assumption beginning after the model's final phase is invalid because it can never affect the analysis.

The words `weak`, `forgeable`, `malleable` and `from` have special meaning only inside assumption brackets. They remain valid constant names elsewhere (§2.4).

4.6.4 Annotation Semantics and Scope

An annotation does not change a term's identity. `SIGN[forgeable](sk, m)` and `SIGN(sk, m)` compare equal and rewrite in the same way. The annotation only enables additional attacker rules.

First, equivalent terms share an annotation. If Alice computes `HASH[weak](m)` and Bob computes `HASH(m)`, both terms are weak because they denote the same value. Repeating a computation without its annotation cannot restore the guarantee.

Second, `forgeable` applies to the primitive under a particular secret rather than to a single term. From `SIGN[forgeable](sk, m)`, the attacker may construct `SIGN(sk, m2)` for any constructible `m2`, even if another call site is unannotated. The assumption does not apply under a different signing key. A loss of authenticity under one key affects every message signed with that key.

At load time, Verifpal prints an `Info` notice for each unannotated occurrence affected through term equivalence or a shared secret. For example, when `sig_b` is written without an annotation but denotes the same term as an annotated `SIGN[forgeable](sk, m)`, the notice states that `sig_b` is analyzed under that assumption too. This makes the annotation's full scope visible to a reviewer.

4.6.5 Reading a Result Under Assumptions

An attack found under a declared assumption is conditional on that assumption. Verifpal prints all active assumptions before every result, whether queries pass or fail. This context must accompany a trace: it is not an unconditional finding about the ideal primitive.

Each affected derivation step also names the assumption on which it depends, as step 2 does in the preceding trace.

4.7 Public Keys and Key Exchange

`PUBKEY(a)` represents the public key derived from private value a. The same constructor supplies keys to `SIGNVERIF`, `PKE_ENC`, `RINGSIGNVERIF`, `KEM_ENCAP` and `DH_KEX`:

• EXAMPLE: PUBLIC KEYS AND KEY EXCHANGE

```
principal Server[
    generates x
    generates y
    gx = PUBKEY(x)
    gy = PUBKEY(y)
    gxy = DH_KEX(gx, y)
    gyx = DH_KEX(gy, x)
  ]
```

Verifpal treats `gxy` and `gyx` as equivalent because $g^{xy} = g^{yx}$. Thus, `DH_KEX(PUBKEY(x), y)` and `DH_KEX(PUBKEY(y), x)` denote the same shared secret.

The second `DH_KEX` argument must be a private value: Verifpal rejects `DH_KEX(PUBKEY(x), PUBKEY(y))`, because two public keys do not suffice to derive the shared secret. The first argument is normally a public key, but it is not restricted to a `PUBKEY` term, since a peer's key usually arrives as a received constant or as the output of a decryption. Only a `DH_KEX` nested inside another `DH_KEX` is rejected in that position, and `PUBKEY` applied to a public key is rejected everywhere. Commutativity applies where the first argument is, or resolves to, a `PUBKEY` term; a first argument that never resolves to one gives the attacker no route to the shared secret from the other side, so a mistyped key can make a query pass that the intended model would fail.

4.7.1 Hybrid Key Exchange

A hybrid exchange combines a classical Diffie–Hellman secret with a post-quantum KEM secret in a key-derivation function. The resulting key remains protected if at least one component remains secret. Both components are ordinary Verifpal terms and can be combined with `HKDF`:

• EXAMPLE: HYBRID KEY EXCHANGE

```
principal Alice[
    generates a, r
    ga = PUBKEY(a)
    ss, ct = KEM_ENCAP(ekb, r)
    dh = DH_KEX(gb, a)
    k = HKDF(dh, ss, nil)
  ]
```

Later phases can expose one component at a time (§2.6). Leaking the Diffie–Hellman private values models a future adversary that can break the classical exchange; leaking the decapsulation key models failure of the KEM. Chapter 8 analyzes both cases.

4.8 The Equational Theory

An *equational theory* defines how symbolic expressions reduce and when terms are equal. It establishes, for example, that decrypting a ciphertext with its matching key yields the plaintext.

The theory follows from the PrimitiveSpec rules and the commutativity of `DH_KEX`. Let E denote $\mathsf{KEM\_ENCAP}(\mathsf{PUBKEY}(dk), r)$, where $E[0]$ is the shared secret and $E[1]$ is the ciphertext. The complete equations are:

- $\mathsf{ASSERT}(x, x) \to nil$ [core rewrite]
- $\mathsf{DEC}(k, \mathsf{ENC}(k, m)) \to m$ [rewrite]
- $\mathsf{AEAD\_DEC}(k, n, \mathsf{AEAD\_ENC}(k, n, m, ad), ad) \to m$ [rewrite]
- $\mathsf{PKE\_DEC}(sk, \mathsf{PKE\_ENC}(\mathsf{PUBKEY}(sk), m)) \to m$ [rewrite]
- $\mathsf{SIGNVERIF}(\mathsf{PUBKEY}(sk), m, \mathsf{SIGN}(sk, m)) \to nil$ [rewrite]
- $\mathsf{RINGSIGNVERIF}(\mathsf{PUBKEY}(sk), \ldots, \mathsf{RINGSIGN}(sk, \ldots)) \to nil$ [rewrite]
- $\mathsf{UNBLIND}(f, m, \mathsf{SIGN}(sk, \mathsf{BLIND}(f, m))) \to \mathsf{SIGN}(sk, m)$ [rewrite]
- $\mathsf{KEM\_DECAP}(dk, E[1]) \to E[0]$ [rewrite]
- $\mathsf{SPLIT}(\mathsf{CONCAT}(a, b, \ldots)) \to (a, b, \ldots)$ [rewrite]
- $\mathsf{ENC}(k, m) \to m$ given k [decompose]
- $\mathsf{AEAD\_ENC}(k, n, m, ad) \to m$ given k, n [decompose]
- $\mathsf{DEC}(k, e) \to e$ given k [decompose]
- $\mathsf{AEAD\_DEC}(k, n, e, ad) \to e$ given k, n [decompose]
- $\mathsf{PKE\_ENC}(\mathsf{PUBKEY}(sk), m) \to m$ given sk [decompose]
- $\mathsf{PKE\_DEC}(sk, e) \to e$ given sk [decompose]
- $\mathsf{BLIND}(f, m) \to m$ given f [decompose]
- $E[1] \to E[0], r$ given dk [decompose]
- $\mathsf{CONCAT}(a, b, \ldots) \to a, b, \ldots$ [concatenation]
- $\mathsf{THRESHOLD\_SPLIT}[t](s)[i_1] + \cdots + \mathsf{THRESHOLD\_SPLIT}[t](s)[i_t] \to s$ for distinct $i_1, \ldots, i_t$ [recompose: t-of-n]
- $\mathsf{THRESHOLD\_JOIN}(s_{i_1}, \ldots, s_{i_t}) \to x$ when the s_i are distinct outputs of one $\mathsf{THRESHOLD\_SPLIT}[t](x)$ [rebuild]
- $\mathsf{THRESHOLD\_JOIN}(p_1, \ldots, p_t) \to \mathsf{SIGN}(x, m)$ for t partials over distinct shares of x, one c and one m [combine]
- $\mathsf{THRESHOLD\_JOIN}(\mathsf{PUBKEY}(s_{i_1}), \ldots, \mathsf{PUBKEY}(s_{i_t})) \to \mathsf{PUBKEY}(x)$ [combine]
- $\mathsf{THRESHOLD\_SIGN}(s, n, c_1, m_1), \mathsf{THRESHOLD\_SIGN}(s, n, c_2, m_2) \to s$ [reuse]
- $\mathsf{AEAD\_ENC}(k, n, m_1, ad_1), \mathsf{AEAD\_ENC}(k, n, m_2, ad_2) \to m_1, m_2$ [reuse]

- $\rightarrow \mathsf{AEAD\_ENC}(k,n,m,ad)$ given m, ad and a reused pair under (k,n) [reuse]
- $\mathsf{DH\_KEX}(\mathsf{PUBKEY}(a),b) = \mathsf{DH\_KEX}(\mathsf{PUBKEY}(b),a)$ [DH commutativity]

The equations distinguish public and private inputs in two places. First, decomposing `PKE_ENC` or `KEM_ENCAP` requires the private value *sk* or *dk*; the public key reveals nothing. The KEM rule also runs on the ciphertext output only: holding the shared secret $E[0]$ reveals neither r nor the ciphertext. Second, decomposing a `DEC`, `AEAD_DEC` or `PKE_DEC` term with the corresponding key reveals the ciphertext input. This rule exposes the input to a principal's decryption when the attacker holds the key, and for `AEAD_DEC` the nonce as well. The two reuse entries are the only rules that consume two held terms rather than one; both need a pair of `AEAD_ENC` terms agreeing on key and nonce, and both hold with no annotation (§4.4).

Verifpal applies these equations from left to right during symbolic evaluation. It performs no equational reasoning beyond the listed rules. The fixed theory bounds the analysis but cannot express user-defined algebraic properties.

During attacker deduction a rewrite is also applied *forward*. When the attacker can obtain every argument of a term that a principal computes, and the rewrite reduces that term, the attacker learns the reduct (§5.2.1). For most rewrites the reduct is reachable anyway, by decomposition for the decryptions and by concatenation for `SPLIT`; `UNBLIND` is the exception, since `SIGN(sk, m)` is neither one of its arguments nor obtainable from the blind signature by any decompose rule. An attacker holding the blinding factor, the message and the blind signature therefore forges the signer's signature on `m` without `sk`.

For an unannotated model, this list is complete, the reuse entries included. A weakening assumption (§4.6) enables an additional attacker rule. **`weak`** reveals the designated input or output without a precondition. **`forgeable`** removes the secret argument required to reconstruct the primitive under that secret (§4.6.4). **`malleable`** allows a held `ENC(k, m)` to support construction of `ENC(k, m2)`. These capabilities do not change term equality or rewriting, so they are not additional equations.

4.9 Quick Reference

Table 4.1 lists each primitive's input and output counts, checkability, deduction rules and supported weakening assumptions.

PRIMITIVE	INPUTS	OUTPUTS	CHECKABLE	DEC.	REW.	W	F	M	NOTES
Core									
ASSERT	2	1	✓		✓				equality of inputs
CONCAT	2–5	1							reveals every input
SPLIT	1	1–5	✓		✓				input must be a CONCAT
Hashing									
HASH	1–5	1				✓			
MAC	2	1					✓		
HKDF	3	1–5							
Public key									
PUBKEY	1	1				✓			not applicable to a public key
DH_KEX	2	1							commutative; arg. 2 is private
Encryption									
ENC	2	1		✓		✓		✓	
DEC	2	1		✓	✓				rewrites, but not checkable
AEAD_ENC	4	1		✓		✓	✓		ad not revealed; one nonce per key
AEAD_DEC	4	1	✓	✓	✓				needs the key and the nonce
PKE_ENC	2	1		✓		✓			decomposes with the private key
PKE_DEC	2	1		✓	✓				rewrites, but not checkable
Signatures									
SIGN	2	1					✓		
SIGNVERIF	3	1	✓		✓				rewrites to nil
RINGSIGN	4	1					✓		
RINGSIGNVERIF	5	1	✓		✓				keys in any order
BLIND	2	1		✓					
UNBLIND	3	1			✓				signature is arg. 3
Key encapsulation									
KEM_ENCAP	2	2		✓		✓			*ct* + *dk* reveal *ss*, *r*; fresh *r*
KEM_DECAP	2	1	✓		✓				no public key in arg. 1
Threshold									
THRESHOLD_SPLIT	1	2–16							[t] required; recomposes from any *t* outputs
THRESHOLD_JOIN	2–16	1							interpolates shares or partials
THRESHOLD_SIGN	4	1					✓		one nonce per share; *t* partials are SIGN

TABLE 4.1 Summary of Verifpal primitives. *Dec.* marks a decomposition rule (what the attacker learns from the term given the right key) and *Rew.* a rewrite rule (how the term reduces during evaluation). Only checkable primitives may take the ? suffix. *W*, *F* and *M* mark the declared weakening assumptions each primitive accepts (§4.6): weak, forgeable and malleable, respectively. A primitive rejects any capability it does not declare.

CHAPTER 5

How Analysis Works

Verifpal searches for executions that contradict a model's queries. It does not prove the absence of such executions, unlike proof-producing verifiers [5]. This chapter describes the analysis algorithm, the validation that precedes every reported attack, and the bounds within which the results hold. The companion paper [10] gives the formal definitions that this chapter summarizes.

5.1 The Verification Pipeline

Verifpal processes a model in four stages:

1. **Parsing.** The `.vp` source becomes an abstract syntax tree containing the attacker, principals, expressions, messages, phases, scenarios and queries.
2. **Expansion.** Scenario expansion creates one copy of the model per declared peer configuration (§2.7), and session expansion then clones every principal and message once per session (§5.6.3). The result is an ordinary model that the user could have written by hand.
3. **Sanity checking.** Verifpal validates the expanded model's structure. Phase numbers must be sequential; referenced principals must be declared; primitive arities and argument restrictions must hold; `?` may appear only on checkable primitives; each authentication query's recipient must actually use the queried value; and every weakening assumption must be supported by its primitive and begin in a phase that the model reaches (§4.6). The validated model is transformed into two internal representations:
   - A *protocol trace*: an immutable global record of each constant's initial value, creator, holders and communication phases.
   - A set of *principal states*: per-principal records of resolved values, network receipt, guards and possible mutation sources.
4. **Analysis.** The engine follows either the passive or active attacker procedure described below.

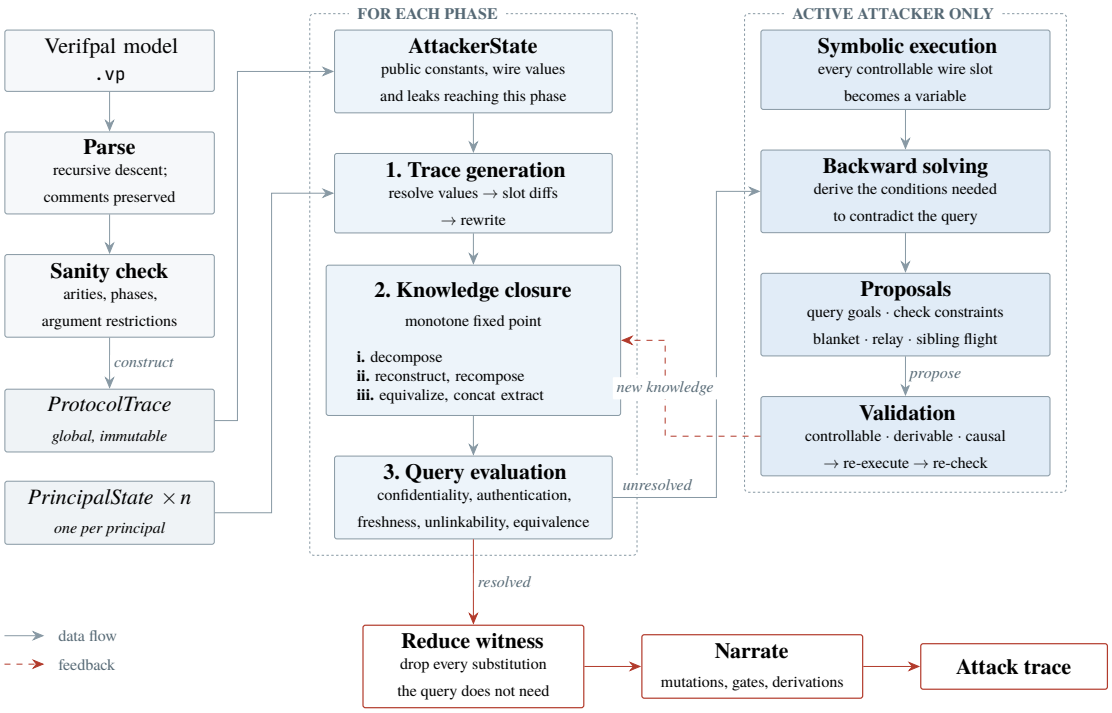


FIGURE 5.1 Verifpal's analysis architecture. Parsing and validation produce an immutable protocol trace and one state per principal. For each phase, the engine initializes attacker knowledge, resolves principal terms and runs deduction to a fixed point before evaluating queries. Under an active attacker, backward search proposes substitutions for unresolved queries. Validation checks every proposal for controllability, derivability and causal availability, then re-executes it; only an executed state can record a contradiction. New knowledge feeds the next analysis round. A confirmed witness is then reduced and rendered as an attack trace.

5.2 Analysis Methodology

The analysis has two main components (Figure 5.1). A *deduction loop* computes all attacker knowledge available from one execution. For an active attacker, a *goal-directed search* works backward from each unresolved query to find network substitutions that may contradict it.

Verifpal evaluates each candidate execution in three ordered stages. *Trace generation* resolves terms and applies rewrites. *Knowledge closure* runs deduction to a fixed point. *Query evaluation* then checks every query against the completed execution and attacker knowledge. Query evaluation never interrupts deduction.

5.2.1 The Deductive Analysis Loop

Given attacker knowledge $\mathcal{V}_A$ and principal state $\mathcal{V}_P$, the deduction loop repeats three rule groups until none derives a new value:

1. **Phase 1: Decompose and break.** For each primitive term in $\mathcal{V}_A$, apply its DECOMPOSE rule when the attacker knows the required inputs, such as a decryption key. The rule opens only terms the attacker holds: knowing a key does not yield the plaintext of a ciphertext the attacker never obtained. For an active `weak` assumption

(§4.6), reveal the protected value without any additional input: hash arguments, a ciphertext's plaintext or a public key's private value.

2. **Phase 2: Reconstruct and Recompose.** For each value in $\mathcal{V}_P$ (the principal's assigned values):
   - *Reconstruct*: add a principal term to $\mathcal{V}_A$ when the attacker knows every component needed to build it. Reconstruction is recursive, so it can derive a nested term such as a Diffie–Hellman secret from a public key and the matching private value. An active `forgeable` assumption removes the secret component from this requirement.
   - *Recompose*: apply a primitive's RECOMPOSE rule when the attacker holds the required subset of outputs, such as any three of five shares from a `THRESHOLD_SPLIT[3]`.
   - *Rewrite forward*: when the principal's term reduces under a rewrite rule and the attacker can obtain every argument of the unreduced term, add the reduct. This is what lets an attacker holding a blinding factor unblind a blind signature (Chapter 4).
3. **Phase 3: Equivalize and Concatenation.** For each value in $\mathcal{V}_A$:
   - *Equivalize*: when a term in $\mathcal{V}_P$ is structurally equivalent to one in $\mathcal{V}_A$, add the principal's representation to $\mathcal{V}_A$. A leak or message that the principal never reached, because a failed check halted it earlier, is not available through this rule.
   - *Concatenation*: extract every component of a known `CONCAT` term.

If any group adds knowledge, the loop restarts at Phase 1. It reaches a *fixed point* when all three groups add nothing. Only then does Verifpal evaluate queries. Every rule records, together with the value it adds, the derivation that produced it; the attack narrator reads these records (§5.4). No rule adds a term merely because of its shape.

The loop terminates because knowledge only grows and every rule produces a subterm, an output projection or a term already present in the principal state. The set of such terms is finite for a given execution, so the iteration reaches its fixed point after finitely many steps.

5.2.2 Passive Attacker Analysis

For each model phase, passive analysis performs these steps:

1. Initialize a fresh attacker state.
2. Add all public constants, currently leaked constants and network values to $\mathcal{V}_A$. A value that a principal would have sent only after a failed check in the honest run is withheld, since that execution never transmitted it.
3. For each principal, resolve all values, perform symbolic rewrites, and run the deductive analysis loop (§5.2.1).

4. Check all queries.

Because a passive attacker cannot modify messages, one honest execution suffices for each phase.

5.2.3 Active Attacker Analysis

An active attacker may substitute terms for any combination of unguarded network values. Forward enumeration of all such combinations would require an arbitrary search budget or depth limit.

Verifpal instead works backward from each unresolved query. It derives the conditions required for a contradiction and asks whether network substitutions can satisfy them. If an attack requires four public-key replacements, the solver derives four independent requirements and combines their bindings. No user-selected search depth or per-principal budget is involved.

The search proceeds, for each phase, as follows:

1. **Passive baseline.** Run passive analysis to identify contradictions that require no message modification.
2. **Search rounds.** Repeat the following until a complete round adds no attacker knowledge and resolves no query. Validating one proposal may reveal knowledge needed by another.

   - A *direct* search runs a *targeted* pass over every principal, pursuing goals derived from the queries, and then a *constructed* pass that offers additional values built from protocol terms already held by the attacker. It repeats both passes until they add no knowledge.
   - If queries remain unresolved, a *refined* search runs the targeted pass again under refinements that keep one selected wire slot at its honest value.
   - If the refined search added knowledge, the direct search runs once more.

   The passes use separate sweeps across all principals. This prevents one principal with many constructible values from delaying targeted work on another.

Symbolic Execution of a Principal

For backward reasoning, Verifpal represents each principal's computation as a function of attacker choices. It replaces every controllable network slot with a variable and then applies ordinary rewriting. An incoming Diffie–Hellman public key becomes `PUBKEY(x)`, not an unconstrained variable. This representation naturally preserves the structure required for a man-in-the-middle substitution.

A slot is uncontrollable when it is guarded, created locally, available only in a later phase or unused by that principal. The exception for guards applies when the forwarding sender originally received the same value unguarded (§2.5); it covers that one hop only.

A value from an earlier phase remains substitutable only with knowledge available in its earliest communication phase (§2.6).

Replacing every controllable slot with a variable loses one option the attacker has: leaving a value alone. A search that sees only the variable cannot reason through the honest value, and some decryption oracles are reachable only that way. The refined search restores this option. It selects a slot whose variable occupies the argument position of a rewrite rule that otherwise cannot fire, restores that one slot to its honest value and repeats the targeted pass. One slot is held honest at a time.

Discharging Goals

A goal is a term that the attacker must hold or cause; it may contain free variables. The solver tries the following rules in order:

- *Known*: the goal is equivalent to a term the attacker holds.
- *Bare variable*: an ordinary free variable stays open; a variable standing for a controllable slot is bound to `nil`.
- *Replay*: the goal unifies with a held term, and every compound term this binds must then be constructible.
- *Wire*: the goal unifies with a symbolic transmitted term. If direct unification fails, the solver tries an *oracle*, which synthesizes an input shape that a recipient's rewrite would turn into the goal, and a *rewrite match*, which reduces a symbolic wire term and unifies the result with the goal.
- *By arguments*: for a primitive goal, solve each argument recursively, except a secret position exempted by an in-force `forgeable` assumption. The Diffie–Hellman exchange is also tried.
- *Decompose*: as the final alternative, open a held term using its registered prerequisites and unify a revealed value with the goal.

Two further rules run while proposals are constructed rather than inside this recursion. *Satisfy check* derives the constraints that make a checked primitive succeed, so that a check the attacker can satisfy does not halt the principal. *Invert* works backward through a rewrite rule to solve for a pattern argument: it constrains the positions the rewrite fixes, fills the others with fresh variables and unifies the rule's output with the target. It therefore proposes only shapes that the rewrite can produce.

Every binding creates a construction obligation. If a wire slot must contain a particular term, the attacker must be able to construct that term. The solver therefore derives an attacker-controlled Diffie–Hellman key rather than assuming a convenient shared secret. If two bindings assign different values to one variable, the solver tries to unify them directly and under the Diffie–Hellman exchange, and discards the pair if both attempts fail.

The oracle rule draws on a finite *oracle basis*: the subterms of the symbolic state and of every term the attacker holds. It cannot introduce arbitrary term shapes. This restriction

is a source of incompleteness that the result envelope does not report, because Verifpal cannot distinguish a goal that is missing from the basis from one that is unsatisfiable.

The solver memoizes goals and cuts cycles. Its goal space is finite, and injected terms may not be nested more deeply than terms computed by the model (§5.3). Primitive argument restrictions are also enforced by unwrapping invalid key-derivation nesting, preventing infinite sequences such as `PUBKEY(PUBKEY(x))`.

Proposal Families

Solved goals become *proposals*: substitutions for one principal's controllable slots. Table 5.1 lists the families in the order they are offered. The targeted and constructed families run during proposal generation. The materialization variants fill positions that a solved shape leaves unconstrained; each proposal that still carries free positions is offered again under each applicable variant. The refinement family runs only in the refined search.

FAMILY	PASS	WHAT IT INSTALLS
blanket	targeted	`nil` at every controllable slot, so a public-key slot becomes `PUBKEY(nil)`. The direct man-in-the-middle substitution.
single slot	targeted	one controllable slot, with every other left honest. Avoids a failure caused by replacing an unrelated field.
check constraint	targeted	the bindings that make a checked primitive succeed.
relay	constructed	one candidate slot over otherwise honest values, so that one field changes while the rest of its message arrives intact.
sibling flight	constructed	every controllable value of another session at once. A replay whose fields must come from one session, such as a key share and its binder.
keyed	materialization	the attacker's own public key at an unconstrained position whose honest term is a public-key derivation.
preserving	materialization	as keyed, and additionally keeps an aligned honest field the attacker already holds. A forgery that changes a key but must keep a neighboring nonce or tag.
distinguish	materialization	distinct public terms at distinct unconstrained positions, offered only while an `equivalence?` query is unresolved, so that grounding cannot hide a divergence between two projections of one forged term.
refinement	refined	one selected slot restored to its honest value, at an argument position of a rewrite that otherwise cannot fire.

TABLE 5.1 Proposal families, in offer order. A family is a way of completing a substitution; every proposal it produces still passes through validation before it can affect a result.

Two details of this order are visible in reported traces. Sibling-flight replays run after the targeted families, so the output prefers a direct man-in-the-middle witness when both

exist. Variables the solver never bound stay free, and validation leaves their slots unchanged: modifying an irrelevant slot could fail an unrelated check before the proposed attack occurs. The distinguish fillers are `nil` and a hash of `nil` at each arity; they are structurally distinct public terms, not fresh nonces, because the attacker cannot invent an atom that does not exist in the model.

Proposals are deduplicated by the concrete state they would install, not by the bindings that produced them, and the same install can be retried in a later round once attacker knowledge has grown.

Proposal and Validation

Backward search cannot record a verdict. It only *proposes* substitutions. Verifpal materializes each proposal as a concrete principal state, executes it forward and checks it against actual attacker knowledge before recording a query contradiction.

The validator checks every proposal independently of the solver, because the facts it needs are not recoverable from the resulting state. A solver defect could, for instance, propose `SIGN(sk, m)` where Bob expects Alice's signature although the attacker does not know `sk`; re-execution alone would install it, pass `SIGNVERIF` and report a forgery. The validator accepts a substitution only when every one of the following holds:

- every bound slot is *controllable* at the current phase (§5.5);
- every installed term is admissible under the argument restrictions after normalization, lies within the term-depth bound (§5.3), and contains no embedded check whose rewrite fails;
- every installed term is *derivable* by the attacker, using knowledge archived no later than the phase of the slot it fills, so that a later leak cannot justify an earlier substitution;
- every installed term is *causally available* at the receive it replaces; and
- the substitution changes at least one slot's value, and the re-execution is defined, meaning that no installed term refers, directly or through other slots, to the slot it fills.

Causal availability is needed because Verifpal collects the attacker's knowledge within a phase without regard to order, so that traffic from one session can be routed into another wherever their blocks appear in the expanded model. Read alone, that would let a check be satisfied with a message the protocol sends only *because* the check already passed. For each attacked receive, Verifpal computes which later transmissions could depend on it, and withholds every known value whose recorded derivation bottoms out in one of them. A term the attacker can build from earlier ingredients remains available even if that exact term never traveled. Chapter 7 shows a substitution that this rule refuses.

Derivability is normalized first. An attacker-supplied `DH_KEX(PUBKEY(a), PUBKEY(b))` normalizes to the honest shared secret `DH_KEX(PUBKEY(a), b)`, whose private operand the attacker must then derive; no rule yields it from two public keys.

Re-execution starts from the principal's pristine state, with every slot the proposal does not name at its honest value, while attacker knowledge is global. Verifpal therefore also applies a *history-coherence* filter: a value learned while executing the creator of a forwarded slot, or the target principal itself, is refused when it disagrees with the honest value that this re-execution relies on, unless the proposal explicitly replaces that slot. Two further filters cover what this one cannot see. A value revealed to a third principal is refused when re-executing its origin under the substitution it was recorded against changes what the target principal holds, and a deduction inside the knowledge closure that combines reads from incompatible executions is refused where it is made, by replaying the union of the substitutions those reads depend on. Every known shape of the combined-execution report is closed by one of the three; the class is not proven empty (§5.6).

Knowledge gained while validating one proposal becomes available in the next round. A phase ends when all queries are resolved or when a complete round adds no knowledge and resolves no query.

5.2.4 Guard Bypass

When a checked primitive fails in a mutated execution, Verifpal determines whether the attacker could have supplied a value that the check accepts. It must hold the *bypass key*, such as the symmetric key of `AEAD_DEC` or the private signing key corresponding to the public key in `SIGNVERIF`, and it must also be able to obtain every other input that the rewrite rule constrains: the nonce and associated data of an `AEAD_DEC`, the signed message of a `SIGNVERIF`, or the ring and message of a `RINGSIGNVERIF`. Holding a leaked key alone does not defeat a check whose other inputs the attacker cannot supply. If both conditions hold, Verifpal constructs a bypass state in which the checked slot holds an attacker-known placeholder and then resolves the principal again.

This process repeats because one bypass may reveal material needed for another. It terminates without an arbitrary bound: each iteration must mark a previously unmarked slot, and every principal has finitely many slots. Verifpal applies the injection before assignments are inlined so that the new value propagates to dependent slots.

For example, substituting `PUBKEY(nil)` for an identity key initially invalidates an honest signature. Because the attacker knows `nil`, it can instead create a signature that verifies under the substituted key. Treating the first failure as final would miss this man-in-the-middle execution.

A bypassed check is not itself a successful use of the value it received (§3.2.2): the check did not rewrite on that value. A later primitive that consumes the placeholder can witness use, and a later check must rewrite normally or be bypassed in turn.

5.3 Preventing State Space Explosion

Symbolic analysis can generate an intractable number of states and terms. Verifpal limits this growth through the following design choices and reports the explicit depth limit when it excludes a proposal:

- **Goal direction.** Search begins at queries and works toward the network (§5.2.3). It never enumerates substitutions unrelated to a query.
- **Memoization.** The solver caches goals and cuts a goal already on the active path, solving each subproblem once per search.
- **A finite term space.** Argument restrictions unwrap invalid nesting and prevent unbounded key-derivation terms. The attacker builds terms only from constants that exist in the model; it cannot mint a fresh nonce of its own, so an attack that requires an attacker-chosen atom distinct from every modeled constant is outside the search.
- **A model-derived depth bound.** An injected term cannot be nested more deeply than the deepest term computed by the protocol. A deeper model therefore receives a deeper bound without a user-selected parameter. The bound is per slot: a slot's cap is the depth of the protocol's deepest term plus the number of layers the receiving principal itself peels off that slot, and the room above the flat cap may be spent only where the excess is carried by the protocol's own subterms. One deep assignment therefore does not loosen the bound for every slot, but a shallow model may still exclude an attack that needs one extra constructor. An attack that requires a still deeper injected term is out of reach. When the bound rejects a proposal, Verifpal prints an `Info` line naming the term, the principal and slot, and the bound:

  ```
  Search declined AEAD_ENC(k, n, HASH(HASH(HASH(HASH(m)))), nil) at Bob's e1:
        it nests deeper than the 4 levels this protocol itself computes.
  ```

  Every passing query in that model then carries the envelope `[search truncated: term depth]` instead of `[search exhausted at k sessions]` (§1.2.1). The attribution is run-wide and therefore conservative: one declined proposal marks every passing query, even those the proposal did not concern.
- **Cached term structure.** Each term caches a structural hash. Knowledge is indexed by this hash, allowing most inequality and membership tests to avoid recursive comparison.
- **Exhaustion detection.** A phase ends as soon as all its queries are resolved, or as soon as a round adds no new attacker knowledge and resolves nothing new.
- **Bounded arity.** A primitive has at most five inputs and five outputs, which bounds rule branching.

The depth bound is what makes the active search terminate. Without it, a protocol that decrypts a value, hashes it and re-encrypts the result under the same key admits an unending replay pump in which each round produces a term one level deeper than the last. Because the bound and the constants are fixed when the model is built, only finitely many states can be installed, and the round loop stops.

One incompleteness source cannot be reported at a specific search step. When the solver needs a complete term rather than one it can assemble, it considers only the finite oracle basis of protocol terms and terms already held by the attacker. An attack requiring another term is out of reach. In that case, an undischarged goal is indistinguishable from an impossible goal.

ProVerif uses over-approximation to reason soundly about unbounded sessions, which can produce inconclusive results or non-termination. Verifpal instead terminates with a concrete witness for each reported attack. The tradeoff is that a passing query is not a proof (§5.6).

5.4 Attack Traces

For each contradicted query, Verifpal prints a numbered sequence of attacker actions followed by the condition that contradicts the query. The trace is the evidence for the verdict.

5.4.1 Witness Reduction

The solver pursues all unresolved queries together and may install the union of their bindings. The state that contradicts one query therefore often contains substitutions that only another query needed. Verifpal reduces the witness before rendering it.

It first collects candidate witnesses: the standard man-in-the-middle substitution for Diffie–Hellman public values, the substitutions recorded for the queried value together with those that its derivation transitively depended on, those substitutions with public keys replaced by `PUBKEY(nil)`, terms shaped to satisfy the recipient's own checks, and honest values from sibling sessions. When a check leaves an argument unconstrained, a candidate uses the honest argument if the attacker holds it, so that a trace describes a forgery over the real payload rather than over `nil`.

Each candidate is probed in a fresh scratch context initialized from the passive baseline, not from end-of-search knowledge, which could otherwise make unnecessary substitutions appear sufficient. The probe installs the candidate's slots in source order and checks, before each one, that the attacker could derive the installed term from what it holds at that point; a candidate that passes every such check is *grounded*. It then re-executes the state and evaluates the query. Verifpal prefers a grounded candidate that bypasses no check, and it then removes substitutions one at a time for as long as the contradiction, the grounding and the absence of bypasses are preserved. The result is a *reduced* witness. It is not necessarily minimal, because a single deletion pass does not test every subset.

Three outcomes are possible, and the trace says which:

- The witness *reproduces* the contradiction and is *grounded*. No label is printed. This is the case for nearly every attack in Verifpal's example corpus.
- The witness reproduces the contradiction but at least one substitution could not be derived from what the attacker held before making it. The trace ends with the following note, and its steps are not a causally ordered execution:

```
• A TRACE THAT IS NOT CAUSALLY ORDERED

Note: this trace reproduces the violation, but at least one
substitution in it could not be derived from what the attacker
holds before making it, so the steps are not a causally ordered
execution.
```

- No candidate reproduces the contradiction. Verifpal reports the substitutions recorded during search instead:

```
• A TRACE THAT IS NOT A MINIMIZED WITNESS

Note: these are the substitutions the search recorded; no subset
of them was confirmed to reproduce the violation on its own, so
this trace is not a minimized witness.
```

Treat the last two cases as *unconfirmed*. The command line still prints `Fail` for them, but the report has not been reproduced as an ordered execution. Reconstruct such an attack by hand before relying on it (§5.6).

5.4.2 Trace Steps

The numbered trace contains eight types of step:

- **Substitutions** group replacements by network message and show each honest value that was replaced.
- **Replays** show a value taken from another run, a session or scenario clone, and delivered in the current one, naming the run it came from.
- **Gates** identify checked primitives that passed with at least one attacker-controlled input. A gate line is printed only after the check succeeded on the installed value.
- **Bypasses** identify failed checks that did not stop the principal because the attacker held the relevant key and could supply an acceptable value (§5.2.3).
- **Derivations** show each attacker deduction in dependency order, including observations, leaks, decompositions and constructions. A derivation performed while walking another session is prefixed with that session. A derivation that depends on a declared weakening assumption names the assumption.
- **Resolutions** state the value to which a source name resolves in the exhibited state; an equivalence trace consists of these.
- **Static values** list the non-fresh inputs behind a freshness failure.
- **Receipts** name the actual sender of a value that an authentication query attributes elsewhere, when no substitution was needed.

Every contradicted query therefore includes either an attack witness or a direct explanation of the violated condition. Traces use model constant names when doing so is unam-

biguous. A replacement step expands the slot it changes to avoid displaying the same name for both the honest and substituted terms, and a multi-output primitive carries its projection, written `|1`, `|2` and so on, wherever a name would otherwise be ambiguous.

The HTML and LaTeX formats render the same witness as a sequence diagram and numbered trace (§1.2.6; §1.2.7). The diagram adds an attacker lane, arrows for interceptions and replays, labels for gates and bypasses, and grouped derivations. Both formats use the same step sequence as terminal output.

5.5 What a Principal Sees

Analysis distinguishes attacker knowledge from each principal's local state. A principal acts on the value it received, without knowing whether the attacker changed it.

Only constants can appear in messages (Figure 1). A constant declared with **`knows`** or **`generates`** resolves to itself. An assigned constant resolves to its primitive term. Primitive arguments may themselves be constants or primitive terms, while primitive outputs are bound to constants.

5.5.1 Provenance

Each known, generated or assigned constant receives a *provenance* record. It stores the *creator*, the latest *sender*, an *attacker_tainted* flag and a *bypassed* flag. If Alice creates `m` and sends it to Bob, Bob's record names Alice as both creator and sender. If the attacker replaces it in transit with a value that differs from the honest one, the sender becomes the attacker and the taint flag is set. A relay that delivers the honest value unchanged keeps its attributed sender.

Authentication queries inspect this record. A contradiction requires a sender other than the claimed one, successful use by the recipient and the absence of a matching run (Chapter 3).

5.5.2 Three Values per Slot

Each principal-state slot records three forms of a term:

- *original*: the term produced by the honest protocol before network modification;
- *pre_rewrite*: the term after attacker modification and before symbolic rewriting; and
- *value*: the canonical term after modification and rewriting.

When resolving a nested expression, Verifpal uses the original term unless the value both arrived over the network and was attacker-tainted. Locally created or unmodified values therefore retain their honest form; a modified received value uses the attacker-supplied form. A defeated check keeps its honest term as provenance, but the principal's later computation sees the attacker-known placeholder that the bypass installed.

This rule models the recipient's actual view. Alice computes honestly with a substituted public key because she cannot observe the substitution itself. Giving a principal access to both the original and modified network values would create spurious authentication behavior.

5.5.3 Controllable Slots

An active attacker may replace a slot of a principal only when the slot is *controllable*: its constant is not `nil`, the principal uses it, it is available in the current phase, and either it arrived over the wire without a guard, or it arrived guarded but the sender itself had received the same constant unguarded. The second clause covers one forwarding hop. A value forwarded over two guarded hops, or a value the sender computed from attacker-chosen input and then sent under a guard, is not controllable at the final recipient. This is a known source of missed attacks (§5.6.3).

An attacker may replace a controllable slot with:

- a protocol value already revealed to the attacker;
- a primitive term constructed from attacker-known values, including a shape required by a recipient rewrite; or
- an attacker-authored value such as `PUBKEY(nil)` or a signature under an attacker-owned key.

The validator re-checks controllability for every proposal, against the principal and phase being validated, because the symbolic execution that decides it during search is untrusted.

5.6 Soundness and Completeness

The distinction between what validation establishes and what a verdict claims determines how Verifpal results should be interpreted.

5.6.1 What Validation Establishes

Every contradiction that Verifpal records was reached in one of two ways: the honest execution contradicted the query, or a substitution passed the validation path of §5.2.3 and the re-executed state contradicted it. The search can only propose; the query evaluators are handed executed states and nothing that could carry a verdict; and there is exactly one write path for results, reached only from those evaluators. The engine forbids unsafe code, and source-level tests pin these call sites and the validation order, so a solver defect cannot manufacture a contradiction by itself.

More precisely, a recorded contradiction means:

- for confidentiality, that the attacker derived a term equivalent to the value the queried slot holds in the exhibited execution, through the supported deduction rules;

- for authentication, that the recipient successfully used the queried value and Verifpal found neither an unchanged delivery nor a matching run for it, or that one sending was accepted by two sessions;
- for freshness, that a principal successfully used a queried value whose resolved term contains no generated constant;
- for unlinkability, that an observable pair of queried values admits one of the link witnesses defined in §3.2.4; and
- for equivalence, that the queried constants resolved to nonequivalent terms in an explored execution.

This is a statement about the path from a proposal to a recorded result. It does not by itself make the query evaluators correct, and it does not make Verifpal's replay model equivalent to an ordinary interleaving in which each session executes exactly once. The next two sections state where the two differ.

5.6.2 Known Sources of False Reports

Two classes of false report are known, and both come from the session model of §5.6.3.

Incompatible histories. Attacker knowledge is global and accumulates across re-executions, and each re-execution of a principal starts from its pristine state. Knowledge learned in one re-execution of Alice can therefore be spent against honest slots that only a different execution of Alice would produce. Consider a model in which Alice either reveals her nonce under an attacker-chosen Diffie–Hellman key or produces Bob's expected ciphertext under Bob's real key, but no single run does both. Left alone, Verifpal would combine the nonce learned in the first execution with the ciphertext of the second and report a forgery that no execution admits. Three filters refuse this (§5.2.3): history coherence at the install, execution agreement for a value revealed to a third principal, and combination coherence inside the knowledge closure. Every known instance is refused and pinned by a regression model, but each filter follows a value's recorded derivation, so a combination reached by a route the records do not describe could still be admitted.

Incomplete matching-run reconstruction. Verifpal exempts a value that a matching run of the declared sender emits (§3.2.2), but it reconstructs only some matching runs. It tries three reconstructions, each confirmed by re-executing the candidate run: installing into the sender the values the attacker handed this recipient, replaying the substitution recorded against the delivered value where that value was read at the sender's own slot, and routing the recipient's own outgoing messages into a sibling run. An honest run that emits the delivered value only under some other routing, such as one whose input was itself replaced by a public value, is reported as an origin failure. These reports are grounded and reproduced, so no label marks them.

A report labeled as not causally ordered or as not a minimized witness (§5.4) is unconfirmed. An unlabeled report can still fall into the second class. For an attack that matters, reconstruct the execution by hand from the trace, as one would for any tool, and cross-check with ProVerif or Tamarin when the protocol lies in a fragment that both can express.

5.6.3 The Session Model

Verifpal's guarantees apply to the executions that it explores. By default, every principal has two concurrent sessions; `--sessions k` changes this count (§1.2.2). Each session has distinct constants declared with **`generates`** and distinct terms derived from them. Values declared with **`knows`** represent long-term state and remain shared. The attacker may route traffic between any sessions.

Before analysis, Verifpal rewrites the model with one copy of every principal and message per session and gives session-local values distinct names. The remaining pipeline analyzes this expansion as an ordinary model, so the same validation applies. Replication exposes attacks that require distinct fresh values from the same role, including cross-session nonce use, challenge forwarding and replay between sessions that share a long-term key. A single-session model cannot represent these values as distinct terms.

The analysis is a *sequential replay* of this expanded model rather than an ordinary interleaving. Each explored execution re-runs one principal from its pristine state, with the proposed substitutions installed, to completion or to a failed check. Knowledge is the only state shared between re-executions, and it is never reset within a phase. This model differs from ordinary bounded concurrency in five ways:

- The session count is bounded. An attack requiring three sessions is outside the default two-session analysis. Increasing `--sessions` widens the bound but does not remove it. Tools such as ProVerif and Tamarin can quantify over unbounded sessions through different analysis strategies, with different termination and precision tradeoffs.
- Re-executing a session reuses that session's generated values rather than drawing new ones. Causal availability (§5.2.3) prevents this reuse from feeding a principal its own later output, but two re-executions of one session still share every nonce.
- Each re-execution starts one principal from its pristine state, and computed dataflow is then followed forward: every message that principal sends is delivered to its recipient as it actually came out, and those recipients run in turn, to a fixed point. If Alice receives an attacker-chosen value, hashes it and sends the hash to Bob under a guard, Bob's re-execution sees the hash Alice actually computed. A forwarded value is not an attacker choice and is not tested for derivability; it is justified by its sender having sent it, and it keeps its sender. A state reached this way answers queries only over values it legitimately holds.
- Knowledge from several re-executions of one session can be combined, which is the first false-report class above.
- Each known value stores one derivation. Causal availability judges a value by that record, so a value that was obtained twice, once early and once late, may be refused on the strength of the late record. This is a further source of missed attacks.

Within these bounds, a passing verdict and its envelope mean that the sequential-replay search was exhausted at the stated session count (§1.2.1). Verifpal cannot identify

a missing attack that requires more concurrent sessions than the selected bound, and `--saturate` reports only an observed interval of stable verdicts (§1.2.3).

5.6.4 The Counterparty Model

Sessions replicate runs with the same peers. Scenarios independently replicate runs with different peers. A protocol may resist concurrent runs with one peer but fail when a principal communicates with two peers.

Without a `scenarios` block, every replicated Alice run uses the same modeled peers. Scenarios bind peer-dependent constants differently and clone the model once per binding (§2.7). This represents attacks in which the attacker is an authorized participant in one run, including identity misbinding, unknown-key-share attacks and Lowe's attack on Needham–Schroeder.

Because expansion produces an ordinary model, the same validation applies. Verifpal marks a scenario corrupt, from the phase of the relevant leak onward, when a bound value depends on a leaked secret (§2.7.4). A corrupt run may stop at a failed checked primitive without causing a model error; an honest run must still complete its checks. Values that a halted run would have sent only after its halt are withheld from the attacker, and are published again only by a re-execution that actually reaches them. Only runs that are honest at the current phase record query violations, while knowledge from every run is retained. This permits analysis of a protocol that correctly rejects a malicious peer without reporting the rejection as an attack.

Scenarios instantiate only the named peers; they do not quantify over an arbitrary population. The model author must anticipate and list the corrupt-peer configuration. Their count is also bounded. A model with two concurrent peers cannot reveal an attack that requires three.

5.6.5 Results Under Declared Assumptions

A model may weaken a primitive (§4.6). A reported attack then holds relative to both the model and the declared assumption. An attack under `SIGN[forgeable]` does not apply to an execution in which the signature remains unforgeable.

A declared assumption only adds attacker capability; it cannot suppress an existing attack. The additional capability participates in deduction and proposal generation, but every proposal is still re-executed and checked as described in §5.2.3, and the validator repeats the capability exemptions in its own derivability test. An assumption therefore cannot alter results for an unannotated model.

Verifpal prints all active assumptions before the verdicts and repeats the relevant assumption in every dependent derivation step.

5.6.6 What Verifpal Does Not Guarantee

Verifpal does *not* provide the following:

- **Verification completeness.** A passing result does not show that no attack exists. Backward search may fail to derive a needed condition, the oracle basis and the depth bound restrict every run, and the equational theory is fixed. Verifpal is an attack-finding tool, not a proof-producing verifier.
- **Unbounded parallel sessions or peers.** Verifpal analyzes a bounded number of sessions per principal: two by default and at most sixteen through `--sessions` (§5.6.3). A `scenarios` block covers only the peers it names (§5.6.4). Attacks requiring more sessions or peers are outside the analysis. Within a session, a principal runs to completion or to a failed check; Verifpal does not interleave partially completed executions of that principal.
- **Observational equivalence.** Verifpal has no behavioral-equivalence analysis. Its `equivalence?` query compares resolved terms, and `unlinkability?` searches one execution for a concrete link witness; neither approximates a property stated as indistinguishability between executions.
- **User-defined algebra.** A protocol whose security depends on structure that the fixed primitive table does not model, such as an XOR-based construction, a pairing-based scheme or invalid-curve behavior, cannot be modeled faithfully, and may be modeled unfaithfully without complaint from the tool.
- **A type discipline.** Every field is a bitstring. Verifpal can therefore report a type-confusion attack that a length- or tag-disciplined implementation excludes. A model may add explicit tags and checks, as Chapter 8 does, but no switch imposes a type discipline on an existing model.
- **Fresh attacker atoms.** The attacker builds terms only from constants that exist in the model. An attack that needs an attacker-chosen atom distinct from every modeled constant is outside the search.
- **Computational soundness.** The Dolev–Yao model idealizes cryptography. Verifpal does not model attacks based on padding oracles, timing, weak random-number generation or other implementation-level behavior.

5.6.7 The Deduction Fixed Point

For one fixed execution, including a fixed set of substitutions, the deduction loop computes every value derivable through the supported decomposition, reconstruction, recomposition and equivalence rules (§5.2.1). At the fixed point, no further application of those rules adds knowledge. The broader incompleteness concerns which executions backward search reaches and which behaviors the fixed symbolic theory represents.

5.6.8 Consequences of Goal-Directed Search

Goal-directed search can find attacks that require several simultaneous substitutions without enumerating combinations. In the Signal model, for example, a man-in-the-middle attack requires three public-key replacements. Backward search derives each replacement as a separate requirement and validates their union. The case studies in

Part III demonstrate this behavior. They do not strengthen a passing verdict into a proof.

Part III applies this method to Signal, Scuttlebutt and a hybrid post-quantum handshake. Each case study states the model assumptions, defines the queries and explains how the verdicts change when an assumption is weakened.

PART III

Protocol Case Studies

CHAPTER 6

The Signal Protocol

The Signal protocol[1] provides end-to-end encryption for the Signal application and has also been adopted by other messaging systems. Its key exchange, key schedule and authentication checks have distinct security roles. The analysis below removes selected controls to identify the properties that depend on them.

6.1 Security Goals

Signal aims to provide message confidentiality, mutual authentication and protection across key compromise. Each participant maintains long-term identity keys and short-lived ephemeral keys. Identity keys authenticate session establishment; ephemeral keys contribute to message-key evolution. This separation supports two additional goals:

- *Forward-secure authenticated key exchange.* Revealing long-term identity keys after session establishment should not reveal earlier message contents.[2]
- *Per-message forward secrecy and post-compromise security.* A state compromise should expose only a bounded range of messages, and later uncompromised ratchet steps should restore protection.[3]

Signal also supports *asynchronous session establishment*. Alice can establish a session and send a message while Bob is offline. Bob publishes pre-generated key material to a server, allowing Alice to complete the initial key exchange without a live response.

6.2 Principals

The model first defines Signal's initial key exchange and then a three-message exchange using the Double Ratchet.

6.2.1 Modeling the Key Exchange

In the complete X3DH exchange (Figure 6.1), Alice derives four Diffie–Hellman secrets:

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

[2] Off-the-Record Messaging also provided forward secrecy before Signal.

[3] The size of the exposed range depends on protocol progress and delivery behavior. Delayed or out-of-order messages can extend the practical compromise window.

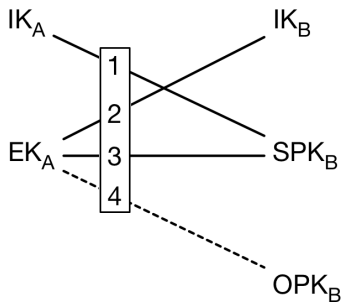


FIGURE 6.1 Signal's X3DH authenticated key exchange. IK_A and IK_B are long-term identity key pairs; EK_A is Alice's ephemeral key pair; and SPK_B and OPK_B are Bob's signed and one-time pre-keys. The exchange derives three required Diffie–Hellman secrets and one optional secret.

1. Alice's long-term private key with Bob's signed pre-key, which Bob generates in advance, signs with his identity key and uploads to the server;
2. Alice's new ephemeral private key with Bob's long-term public key;
3. Alice's ephemeral private key with Bob's signed pre-key; and
4. Alice's ephemeral private key with Bob's one-time pre-key, which Bob generates in advance and uploads without an individual signature.[4]

The four values are combined into a *master secret*. Alice can include an encrypted message with the handshake because every value she needs is available from the server. The model begins with Alice's identity key and Bob's identity, signed pre-key and one-time pre-key:

• SIGNAL: INITIALIZING ALICE

```
attacker[active]
principal Alice[
    knows private alongterm
    galongterm = PUBKEY(alongterm)
  ]
```

• SIGNAL: INITIALIZING BOB

```
principal Bob[
    knows private blongterm, bs
    generates bo
    gblongterm = PUBKEY(blongterm)
    gbs = PUBKEY(bs)
    gbo = PUBKEY(bo)
    gbssig = SIGN(blongterm, gbs)
  ]
```

The model declares the signed pre-key `bs` with **`knows`** because it is shared across sessions.

[4] A signed pre-key serves many sessions before rotation. A one-time pre-key is consumed by one session.

It declares the one-time pre-key `bo` with **`generates`** because each session consumes a distinct value. These declarations reproduce the two key lifetimes under Verifpal's session model (§5.6.3).

Alice receives Bob's public bundle, verifies the signed pre-key and derives `amaster`:

• SIGNAL: ALICE INITIATES SESSION WITH BOB

```
Bob → Alice: [gblongterm], gbssig, gbs, gbo
principal Alice[
    _ = SIGNVERIF(gblongterm, gbs, gbssig)?
    generates ae1
    gae1 = PUBKEY(ae1)
    amaster = HASH(DH_KEX(gbs, alongterm), DH_KEX(gblongterm, ae1), DH_KEX(gbs,
        ae1), DH_KEX(gbo, ae1))
  ]
```

The checked `SIGNVERIF` appears before any use of `gbs`. A failed signature therefore stops Alice before key derivation, as required by X3DH. The anonymous output _ indicates that only the success or failure matters.

6.2.2 Modeling Messages and the Double Ratchet

After the handshake, authentication derives from the master secret, while evolving ephemeral secrets refresh message confidentiality. Signal's Double Ratchet combines these sources as follows:

• SIGNAL: ALICE ENCRYPTS MESSAGE 1 TO BOB

```
principal Alice[
    generates m1, ae2, n_e1
    gae2 = PUBKEY(ae2)
    akshared1 = DH_KEX(gbs, ae2)
    arkab1, ackab1 = HKDF(amaster, akshared1, nil)
    akenc1 = HKDF(nil, MAC(ackab1, nil), nil)
    e1 = AEAD_ENC(akenc1, n_e1, m1,
      HASH(galongterm, gblongterm, gae2))
  ]
Alice → Bob: [galongterm], gae1, gae2, n_e1, e1
```

Alice generates the ephemeral key pair `(ae2, gae2)` and derives `akshared1`. The root-key step passes the previous root value as the `HKDF` salt and the fresh Diffie–Hellman secret as input key material. It produces a new root key `arkab1` and chain key `ackab1`. A second derivation produces the message key `akenc1`. The message key thus depends on both the authenticated master secret and new ephemeral material.

The guards on `gblongterm` and `galongterm` model mutual pre-authentication of the identity keys, such as out-of-band comparison of Signal safety numbers. The values remain visible but cannot be replaced in transit.

Alice encrypts `m1` as `e1`. Its associated data contains both identity keys and the header's ratchet public key. This binds the ciphertext to the session. The identity keys use the same ordering in both message directions; the ratchet key identifies the current header. Each message also carries its own generated nonce, which travels beside the ciphertext because Bob needs it to decrypt (§4.4). Signal derives that nonce from the message key rather than sending it; modeling it as a fresh public value is the conservative choice, since it gives the attacker one more thing to see and nothing to guess.

Bob derives the same master secret and message key, then checks authenticated decryption:

• SIGNAL: BOB DERIVES SHARED MASTER SECRET

```
principal Bob[
    bmaster = HASH(DH_KEX(galongterm, bs), DH_KEX(gae1, blongterm), DH_KEX(gae1,
        bs), DH_KEX(gae1, bo))
  ]
```

• SIGNAL: BOB DECRYPTS ALICE'S MESSAGE 1

```
principal Bob[
    bkshared1 = DH_KEX(gae2, bs)
    brkab1, bckab1 = HKDF(bmaster, bkshared1, nil)
    bkenc1 = HKDF(nil, MAC(bckab1, nil), nil)
    m1_d = AEAD_DEC(bkenc1, n_e1, e1,
      HASH(galongterm, gblongterm, gae2))?
  ]
```

Bob uses the opposite half of each key pair. Diffie–Hellman commutativity makes his four inputs equivalent to Alice's (§4.7).

For the reply, Bob generates a new ratchet key pair and mixes its Diffie–Hellman secret with the previous root key. He encrypts `m2` under the resulting message key:

• SIGNAL: BOB ENCRYPTS MESSAGE 2 TO ALICE

```
principal Bob[
    generates m2, be, n_e2
    gbe = PUBKEY(be)
    bkshared2 = DH_KEX(gae2, be)
    brkba2, bckba2 = HKDF(brkab1, bkshared2, nil)
    bkenc2 = HKDF(nil, MAC(bckba2, nil), nil)
    e2 = AEAD_ENC(bkenc2, n_e2, m2,
      HASH(galongterm, gblongterm, gbe))
  ]
Bob → Alice: gbe, n_e2, e2
```

After decrypting Bob's reply, Alice advances the ratchet again and sends `m3`:

• SIGNAL: ALICE DECRYPTS MESSAGE 2

```
principal Alice[
    akshared2 = DH_KEX(gbe, ae2)
    arkba2, ackba2 = HKDF(arkab1, akshared2, nil)
    akenc2 = HKDF(nil, MAC(ackba2, nil), nil)
    m2_d = AEAD_DEC(akenc2, n_e2, e2,
      HASH(galongterm, gblongterm, gbe))?
  ]
```

• SIGNAL: ALICE ENCRYPTS MESSAGE 3 TO BOB

```
principal Alice[
    generates m3, ae3, n_e3
    gae3 = PUBKEY(ae3)
    akshared3 = DH_KEX(gbe, ae3)
    arkab3, ackab3 = HKDF(arkba2, akshared3, nil)
    akenc3 = HKDF(nil, MAC(ackab3, nil), nil)
    e3 = AEAD_ENC(akenc3, n_e3, m3,
      HASH(galongterm, gblongterm, gae3))
  ]
Alice → Bob: gae3, n_e3, e3
```

• SIGNAL: BOB DECRYPTS MESSAGE 3

```
principal Bob[
    bkshared3 = DH_KEX(gae3, be)
    brkab3, bckab3 = HKDF(brkba2, bkshared3, nil)
    bkenc3 = HKDF(nil, MAC(bckab3, nil), nil)
    m3_d = AEAD_DEC(bkenc3, n_e3, e3,
      HASH(galongterm, gblongterm, gae3))?
  ]
```

After the exchange, phase 1 leaks both long-term private keys but no ephemeral key. This models a later compromise of both devices (§2.6):

• SIGNAL: LONG-TERM PRIVATE KEY LEAKAGE IN SUBSEQUENT PHASE

```
phase[1]

principal Alice[leaks alongterm]
principal Bob[leaks blongterm]
```

The completed model contains an authenticated key exchange, three ratcheted messages and a later compromise of both identity keys.

6.3 Queries and Analysis

The queries test confidentiality for all three plaintexts and authentication for each ciphertext in its sending direction:

• SIGNAL: MESSAGE QUERIES

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

The initial model produces these verdicts:

• SIGNAL: INITIAL ANALYSIS RESULTS

```
Verifpal ◆ Verification completed for 'signal.vp' at 09:38:01 AM in 621ms.

Pass ✓ confidentiality? m1  [search exhausted at 2 sessions]
Pass ✓ authentication? Alice → Bob: e1  [search exhausted at 2 sessions]
Pass ✓ confidentiality? m2  [search exhausted at 2 sessions]
Pass ✓ authentication? Bob → Alice: e2  [search exhausted at 2 sessions]
Pass ✓ confidentiality? m3  [search exhausted at 2 sessions]
Pass ✓ authentication? Alice → Bob: e3  [search exhausted at 2 sessions]

Pass ✓ All 6 queries pass.
```

Verifpal finds no contradiction under the model's assumptions, and every envelope records that the two-session search was exhausted. This result is consistent with earlier formal analyses of Signal [20, 21]. The scope matters: both parties pre-authenticate identity keys, and Alice stops if Bob's signed pre-key fails verification. The confidentiality queries still pass after both identity private keys leak in phase 1, which is the modeled forward-secrecy result.

The first variant removes `?` from Alice's `SIGNVERIF`. Alice now continues after an invalid signed pre-key:

• SIGNAL: RESULTS WITH SIGNVERIF UNCHECKED

```
Fail × confidentiality? m1
Attack trace:
| 1. Attacker constructs PUBKEY(nil).
| 2. Attacker replaces gbs, gbo (sent by Bob to Alice) with
| PUBKEY(nil), PUBKEY(nil). (gbs was PUBKEY(bs);
| gbo was PUBKEY(bo))
| 3. Attacker observes e1 on the wire.
| 4. Attacker is handed alongterm by a leaks declaration.
| 5. Attacker constructs DH_KEX(PUBKEY(nil), alongterm).
| 6. Attacker observes gae1 on the wire.
| 7. Attacker is handed blongterm by a leaks declaration.
| 8. Attacker constructs DH_KEX(gae1, blongterm).
| 9. Attacker constructs DH_KEX(gae1, nil).
| 10. Attacker constructs amaster.
| 11. Attacker observes gae2 on the wire.
| 12. Attacker constructs akshared1.
| 13. Attacker constructs ackab1.
| 14. Attacker constructs MAC(ackab1, nil).
| 15. Attacker constructs akenc1.
| 16. Attacker observes n_e1 on the wire.
| 17. Attacker opens e1 with akenc1 and n_e1, obtaining m1.
> m1 (m1) is obtained by Attacker.
Pass ✓ authentication? Alice→ Bob: e1  [search exhausted at 2 sessions]
Pass ✓ confidentiality? m2  [search exhausted at 2 sessions]
Pass ✓ authentication? Bob→ Alice: e2  [search exhausted at 2 sessions]
Pass ✓ confidentiality? m3  [search exhausted at 2 sessions]
Pass ✓ authentication? Alice→ Bob: e3  [search exhausted at 2 sessions]

Fail × 1 of 6 queries failed.
```

The missing check exposes `m1` after the phase 1 compromise. Working backward from the final step, the attacker derives `akenc1` from `ackab1` (steps 12–15), which depends on `amaster` (step 10) and its four Diffie–Hellman inputs. By replacing `gbs` and `gbo` with `PUBKEY(nil)` in step 2, the attacker makes the first input computable from the leaked `alongterm` (steps 4–5) and collapses the third and fourth inputs to `DH_KEX(gae1, nil)` (step 9). The phase 1 leak of `blongterm` supplies the second input (steps 7–8). Without a checked signature verification, Alice accepts the substituted pre-keys. The trace names `amaster`, `akshared1` and `ackab1` by their model names because the attacker's reconstruction resolves to exactly those terms.

The later plaintexts remain confidential. Their ratchet steps incorporate ephemeral private values that the attacker never obtains. The variant therefore identifies both the property lost and the interval affected by the missing check.

The second variant also removes the guard from Bob's identity public key, modeling a session in which Alice did not pre-authenticate that key:

• SIGNAL: RESULTS WITH BOB'S IDENTITY KEY UNGUARDED

```
Fail × confidentiality? m1
Attack trace:
| 1. Attacker constructs PUBKEY(nil).
| 2. Attacker replaces gblongterm, gbs, gbo (sent by Bob to
| Alice) with PUBKEY(nil), PUBKEY(nil), PUBKEY(nil).
| (gblongterm was PUBKEY(blongterm); gbs was PUBKEY(bs);
| gbo was PUBKEY(bo))
| 3. Attacker observes e1 on the wire.
| 4. Attacker observes galongterm on the wire.
| 5. Attacker constructs DH_KEX(galongterm, nil).
| 6. Attacker observes gae1 on the wire.
| 7. Attacker constructs DH_KEX(gae1, nil).
| 8. Attacker constructs amaster.
| 9. Attacker observes gae2 on the wire.
| 10. Attacker constructs akshared1.
| 11. Attacker constructs ackab1.
| 12. Attacker constructs MAC(ackab1, nil).
| 13. Attacker constructs akenc1.
| 14. Attacker observes n_e1 on the wire.
| 15. Attacker opens e1 with akenc1 and n_e1, obtaining m1.
> m1 (m1) is obtained by Attacker.
Pass ✓ authentication? Alice→ Bob: e1  [search exhausted at 2 sessions]
Pass ✓ confidentiality? m2  [search exhausted at 2 sessions]
```

• SIGNAL: RESULTS WITH BOB'S IDENTITY KEY UNGUARDED (CONT.)

```
Fail × authentication? Bob → Alice: e2
Attack trace:
| 1. Attacker constructs PUBKEY(nil).
| 2. Attacker observes galongterm on the wire.
| 3. Attacker constructs DH_KEX(galongterm, nil).
| 4. Attacker observes gae1 on the wire.
| 5. Attacker constructs DH_KEX(gae1, nil).
| 6. Attacker constructs HASH(DH_KEX(galongterm, nil),
| DH_KEX(gae1, nil), DH_KEX(gae1, nil), DH_KEX(gae1, nil)).
| 7. Attacker observes gae2 on the wire.
| 8. Attacker constructs DH_KEX(gae2, nil).
| 9. Attacker constructs HKDF(amaster, akshared1, nil)|1.
| 10. Attacker constructs HKDF(arkab1, akshared1, nil)|2.
| 11. Attacker constructs MAC(ackba2, nil).
| 12. Attacker constructs HKDF(nil, MAC(ackba2, nil), nil)|1.
| 13. Attacker observes n_e2 on the wire.
| 14. Attacker replaces gblongterm, gbs, gbo (sent by Bob to
| Alice) with PUBKEY(nil), PUBKEY(nil), PUBKEY(nil).
| (gblongterm was PUBKEY(blongterm); gbs was PUBKEY(bs);
| gbo was PUBKEY(bo))
| 15. Attacker replaces gbe, e2 (sent by Bob to Alice) with
| PUBKEY(nil), AEAD_ENC(akenc2, n_e2, nil, HASH(galongterm,
| PUBKEY(nil), PUBKEY(nil))). (gbe was PUBKEY(be); e2 was
| AEAD_ENC(bkenc2, n_e2, m2, HASH(galongterm, gblongterm,
| gbe)))
| 16. Alice's AEAD_DEC(akenc2, n_e2, AEAD_ENC(akenc2, n_e2,
| nil, HASH(galongterm, PUBKEY(nil), PUBKEY(nil))),
| HASH(galongterm, PUBKEY(nil), PUBKEY(nil)))? passes —
| the attacker controls one of its inputs.
> e2 (AEAD_ENC(akenc2, n_e2, nil, HASH(galongterm,
PUBKEY(nil), PUBKEY(nil)))), sent by Attacker and not by
Bob, is successfully used in AEAD_DEC(akenc2, n_e2, e2,
HASH(galongterm, gblongterm, gbe))? within Alice's state.
Fail × confidentiality? m3
(25-step trace omitted: the same forgery of Bob's reply,
after which the attacker follows Alice's next ratchet step
to akenc3 and opens e3)
Pass ✓ authentication? Alice → Bob: e3  [search exhausted at 2 sessions]

Fail × 3 of 6 queries failed.
```

Step 2 of the first trace replaces three public keys at once. Goal-directed search derives each replacement independently from the requirement to reconstruct `amaster`; it does not enumerate three-element substitution sets (§5.2.3). With every key on Bob's side under attacker control, `amaster` needs no leaked value at all: the attacker computes all four Diffie–Hellman inputs from Alice's public keys and `nil` (steps 4–8).

The second trace impersonates Bob. The attacker rebuilds Alice's ratchet state from the substituted keys (steps 3–12), forges a ciphertext under the message key that Alice will derive, with associated data that names the substituted identity and ratchet keys (step 15),

and Alice's checked decryption accepts it (step 16). The nonce needs no forging: Bob published `n_e2` beside the ciphertext, so the attacker reads it off the wire in step 13 and seals its own plaintext under the same one. Because `gbs` and `gbe` are both replaced by `PUBKEY(nil)`, Alice's two ratchet secrets collapse to the same term `DH_KEX(gae2, nil)`, which the trace names `akshared1` in step 10. The omitted 23-step trace for `m3` performs the same forgery and then follows Alice's next ratchet step, which now depends only on attacker-known values.

The query **`authentication?`** `Alice` → `Bob: e1` still passes because Alice's identity key `galongterm` remains guarded. The attacker can impersonate Bob to Alice but not Alice to Bob under this variant.

Confidentiality fails for `m1` and `m3`, while `m2` remains protected. Authentication fails only for Bob-to-Alice traffic. The remaining guard therefore preserves one direction of authentication and the properties that depend on it.

In the initial model, both identity keys are guarded, Alice checks the signed pre-key and both identity private keys leak only after the exchange. Verifpal finds no attack on the six queries under those assumptions.

The variants make the dependency structure explicit. Removing the signature check identifies its contribution to first-message forward secrecy. Removing one identity-key guard identifies the direction of authentication that pre-authentication protects. A passing base model alone would not reveal either dependency.

CHAPTER 7

The Scuttlebutt Handshake

Scuttlebutt[1] is a decentralized communication protocol. This chapter analyzes its authenticated key exchange rather than the full application protocol.

7.1 Security Goals

The model examines four handshake goals stated in the protocol documentation:

- *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 know a key identifying the Scuttlebutt network, but a network attacker should not learn it from the handshake.
- *Forward secrecy.* A later compromise of a long-term private key should not decrypt previously recorded handshakes.

7.2 Principals

Each principal has a long-term identity key pair and a fresh ephemeral key pair:

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

• DECLARING NEW PRINCIPALS: ALICE AND BOB

```
principal Alice[
    knows private n
    knows private longTermA
    generates ephemeralA
    longTermAPub = PUBKEY(longTermA)
    ephemeralAPub = PUBKEY(ephemeralA)
  ]
principal Bob[
    knows private n
    knows private longTermB
    generates ephemeralB
    longTermBPub = PUBKEY(longTermB)
    ephemeralBPub = PUBKEY(ephemeralB)
  ]
Bob → Alice: [longTermBPub]
```

The network identifier `n` is initially declared private and pre-shared. This assumption excludes malicious network members, because every member must know `n`. A later variant makes `n` public to measure the properties that depend on its secrecy. The example distributed with Verifpal, `examples/messaging/scuttlebutt.vp`, uses the public declaration and therefore includes attacks by network members.

The exchange spans two round trips. Alice and Bob first exchange ephemeral public keys and MACs:

• 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
```

Each `MAC` binds an ephemeral public key to `n`. A key and MAC from another network will not pass the checked assertion. This mechanism does not distinguish sessions within the same network.

Alice derives one master secret to protect her identity and a second to protect the authenticated session and its messages:

• SCUTTLEBUTT: ALICE GENERATES SESSION SECRETS

```
principal Alice[
    nMacBobValid = ASSERT(MAC(n, ephemeralBPub), nMacBob)?
    ephemeralSecretAlice = DH_KEX(ephemeralBPub, ephemeralA)
    longTermSecretAlice = DH_KEX(longTermBPub, ephemeralA)
    masterSecret1Alice = HASH(n, ephemeralSecretAlice, longTermSecretAlice)
    sig1Alice = SIGN(longTermA, HASH(n, longTermBPub, ephemeralSecretAlice))
    generates n1, n2
    secretBox1Alice = AEAD_ENC(masterSecret1Alice, n1, sig1Alice, nil)
    secretBox2Alice = AEAD_ENC(masterSecret1Alice, n2, longTermAPub, nil)
    longEphemeralSecretAlice = DH_KEX(ephemeralBPub, longTermA)
    masterSecret2Alice = HASH(n, ephemeralSecretAlice, longTermSecretAlice,
        longEphemeralSecretAlice)
  ]
Alice → Bob: n1, secretBox1Alice, n2, secretBox2Alice
```

Bob reconstructs the first master secret, decrypts Alice's signature and identity key, verifies the signature, and derives the remaining Diffie–Hellman input:

• SCUTTLEBUTT: BOB GENERATES SESSION SECRETS

```
principal Bob[
    ephemeralSecretBob = DH_KEX(ephemeralAPub, ephemeralB)
    longTermSecretBob = DH_KEX(ephemeralAPub, longTermB)
    masterSecret1Bob = HASH(n, ephemeralSecretBob, longTermSecretBob)
    sig1Bob = AEAD_DEC(masterSecret1Bob, n1, secretBox1Alice, nil)?
    longTermAPub_Bob = AEAD_DEC(masterSecret1Bob, n2, secretBox2Alice, nil)?
    sig1Valid = SIGNVERIF(longTermAPub_Bob, HASH(n, longTermBPub,
        ephemeralSecretBob), sig1Bob)?
    longEphemeralSecretBob = DH_KEX(longTermAPub_Bob, ephemeralB)
  ]
```

Bob signs the transcript and encrypts the signature under the second master secret:

• SCUTTLEBUTT: BOB SIGNS SESSION TRANSCRIPT

```
principal Bob[
    sig2Bob = SIGN(longTermB, HASH(n, sig1Bob, longTermAPub_Bob,
        ephemeralSecretBob))
    masterSecret2Bob = HASH(n, ephemeralSecretBob, longTermSecretBob,
        longEphemeralSecretBob)
    generates n3
    secretBox1Bob = AEAD_ENC(masterSecret2Bob, n3, sig2Bob, nil)
  ]
Bob → Alice: n3, secretBox1Bob
```

After Alice verifies Bob's transcript signature, the model sends one encrypted message in each direction:

• SCUTTLEBUTT: ALICE ENCRYPTS AND SENDS MESSAGE TO BOB

```
principal Alice[
    knows private m1
    sig2Alice = AEAD_DEC(masterSecret2Alice, n3, secretBox1Bob, nil)?
    sig2Valid = SIGNVERIF(longTermBPub, HASH(n, sig1Alice, longTermAPub,
        ephemeralSecretAlice), sig2Alice)?
    generates n4
    secretBoxM1Alice = AEAD_ENC(masterSecret2Alice, n4, m1, nil)
  ]
Alice → Bob: n4, secretBoxM1Alice
```

• SCUTTLEBUTT: BOB RECEIVES AND DECRYPTS MESSAGE FROM ALICE

```
principal Bob[
    knows private m2
    m1Bob = AEAD_DEC(masterSecret2Bob, n4, secretBoxM1Alice, nil)?
    generates n5
    secretBoxM2Bob = AEAD_ENC(masterSecret2Bob, n5, m2, nil)
  ]
```

• SCUTTLEBUTT: BOB ENCRYPTS AND SENDS MESSAGE TO ALICE

```
Bob → Alice: n5, secretBoxM2Bob
principal Alice [
    m2Alice = AEAD_DEC(masterSecret2Alice, n5, secretBoxM2Bob, nil)?
  ]
```

The model now contains the complete handshake and one application message in each direction.

7.3 Queries and Analysis

In model terms, the four goals require that the attacker cannot:

- obtain Alice's identity public key `longTermAPub`;
- obtain `m1` or `m2`;
- obtain the network identifier `n`; or
- recover earlier messages after a later long-term-key compromise.

The model adds authentication queries for every encrypted box and an equivalence query for the final master secrets:

• SCUTTLEBUTT: CONFIDENTIALITY, AUTHENTICATION AND EQUIVALENCE QUERIES

```
queries[
    confidentiality? n
    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
    equivalence? masterSecret2Alice, masterSecret2Bob
  ]
```

The equivalence query is not one of the four documented goals, but a key exchange must also give both participants the same key. Confidentiality and authentication queries do not directly test this agreement.

The initial model produces these verdicts:

• SCUTTLEBUTT: INITIAL RESULTS

```
Pass ✓ confidentiality? n  [search exhausted at 2 sessions]
Pass ✓ confidentiality? m1  [search exhausted at 2 sessions]
Pass ✓ confidentiality? m2  [search exhausted at 2 sessions]
Pass ✓ confidentiality? longtermapub  [search exhausted at 2 sessions]
Pass ✓ authentication? Alice → Bob: secretbox1alice  [search exhausted at 2
    sessions]
Pass ✓ authentication? Alice → Bob: secretbox2alice  [search exhausted at 2
    sessions]
Fail × authentication? Bob → Alice: secretbox1bob
Fail × authentication? Alice → Bob: secretboxm1alice
Fail × authentication? Bob → Alice: secretboxm2bob
Fail × equivalence? mastersecret2alice, mastersecret2bob

Fail × 4 of 10 queries failed.
```

Verifpal finds no confidentiality attack. It contradicts three of the five authentication queries and the equivalence query, and the four failures split into two groups. Two of them occur within a single run of each participant, and `--sessions 1` reproduces exactly those two (§1.2.2). The other two need the second concurrent session that Verifpal analyzes by default, and §7.3.2 takes them separately.

7.3.1 Cross-Position Ciphertext Substitution

The two single-run traces are short, because the attacker forges nothing:

• SCUTTLEBUTT: THE TWO AUTHENTICATION TRACES

```
Fail × authentication? Alice → Bob: secretboxm1alice
Attack trace:
| 1. Attacker observes n3 on the wire.
| 2. Attacker observes secretbox1bob on the wire.
| 3. Attacker replaces n4, secretboxm1alice (sent by Alice to
| Bob) with n3, secretbox1bob. (secretboxm1alice was
| AEAD_ENC(mastersecret2alice, n4, m1, nil))
| 4. Bob's AEAD_DEC(mastersecret2bob, n3, secretbox1bob,
| nil)? passes — the attacker controls one of its inputs.
| 5. Attacker observes ephemeralapub on the wire.
| 6. Attacker observes ephemeralbpub on the wire.
| 7. Attacker observes longtermbpub on the wire.
| 8. Attacker observes n2 on the wire.
| 9. Attacker observes secretbox2alice on the wire.
> secretboxm1alice (secretbox1bob), sent by Attacker and not
by Alice, is successfully used in AEAD_DEC(mastersecret2bob,
n4, secretboxm1alice, nil)? within Bob's state.
Fail × authentication? Bob → Alice: secretboxm2bob
Attack trace:
| 1. Attacker observes n3 on the wire.
| 2. Attacker observes secretbox1bob on the wire.
| 3. Attacker replaces n5, secretboxm2bob (sent by Bob to
| Alice) with n3, secretbox1bob. (secretboxm2bob was
| AEAD_ENC(mastersecret2bob, n5, m2, nil))
| 4. Alice's AEAD_DEC(mastersecret2alice, n3, secretbox1bob,
| nil)? passes — the attacker controls one of its inputs.
| 5. Alice's SIGNVERIF(longtermbpub, HASH(n, sig1alice,
| longtermapub, ephemeralsecretalice), sig2alice)? passes —
| the attacker controls one of its inputs.
| 6. Attacker observes ephemeralbpub on the wire.
| 7. Attacker observes ephemeralapub on the wire.
| 8. Attacker observes longtermbpub on the wire.
> secretboxm2bob (secretbox1bob), sent by Attacker and not by
Bob, is successfully used in AEAD_DEC(mastersecret2alice,
n5, secretboxm2bob, nil)? within Alice's state.
```

The parenthetical term in each conclusion is the attacker's replacement. In the first trace, Bob accepts his own signature box, reflected back to him, in place of Alice's message box. In the second, Alice accepts Bob's signature box a second time, in place of his message box. Both replacements are authentic ciphertexts produced by the protocol, and step 4 of each trace is the checked decryption that accepts them.

Each substitution moves two values, because a ciphertext no longer travels alone: the box is delivered together with the nonce it was sealed under, and `AEAD_DEC` needs both (§4.4). This does not make the attack harder. The nonce is a public wire value, so carrying it along costs the attacker nothing, and it is a further sign of what is missing: the pair is self-consistent, and the protocol never says which position of the transcript it belongs to.

Both substitutions succeed because the model encrypts every box after the first ex-

change under the same key and the same empty associated data. In an honest run, `masterSecret2Alice` and `masterSecret2Bob` agree, but a ciphertext contains no domain separator for its direction or transcript position. Any valid box under that key can therefore decrypt in another box's slot. This is a *transcript-separation* failure: the cryptographic operations remain ideal and the key remains secret, yet a valid ciphertext has the wrong protocol meaning. The deployed Scuttlebutt construction derives direction-specific keys and numbers its boxes; this simplified model does neither.

Within one run, the third Bob-to-Alice box, `secretBox1Bob`, has only one candidate replacement: `secretBoxM2Bob`, the other ciphertext under Bob's second master secret. Bob sends that box two messages later, only after Alice has accepted the box under attack, and Verifpal's validator requires every injected value to be available before the receive it replaces (§5.2.3). At `--sessions 1` the substitution is therefore refused and the query passes, on grounds of causal order rather than construction. The next section shows what happens when a second run is available to supply the same box in time.

7.3.2 What the Second Session Adds

The remaining two failures are the ones that need Verifpal's default of two concurrent sessions per principal (§5.6.3). Both come from the same source: nothing in a hello ties an ephemeral key to a particular run.

Each `MAC` binds an ephemeral public key to `n`, not to a session, so the attacker may deliver Bob's hello from his second session into Alice's first. Alice's checked `ASSERT` accepts it, and she derives her secrets from an ephemeral key that Bob is not using in the run she believes she is in.

That alone would only abort the handshake, and at one session it does: Alice's first master secret then differs from Bob's, his checked decryption of her signature box fails, and he halts before computing `masterSecret2Bob`. Verifpal does not compare a value against one the other party never reached (§3.2.5), so the equivalence query passes there.

With a second session the attacker has somewhere to route the rest of the flight. It crosses the two runs: Alice's hello is answered with the other Bob's ephemeral, and Bob is given the other Alice's hello and her whole sealed flight, nonces included. Both sides now complete every check, because every value they receive is an authentic value produced by an honest run. They finish holding different master secrets, and the equivalence query is contradicted. The same routing supplies `secretBox1Bob`'s missing replacement: the second run's `secretBoxM2Bob` exists early enough to be delivered in place of the first run's signature box, so the causal-order argument above no longer applies and that query fails as well.

Neither failure forges anything or breaks a key. They are what an exchange loses when its ephemeral keys are authenticated as belonging to the network rather than to a run.

7.3.3 Unguarding Bob's Long-Term Key

The next variant removes the guard from `longTermBPub`:

```
• SCUTTLEBUTT: RESULTS WITH BOB'S LONG-TERM KEY UNGUARDED

Fail × 4 of 10 queries failed.
```

No verdict changes. In this model, the private network identifier `n` acts as a pre-shared symmetric key.[2] The attacker cannot construct valid hello MACs without `n`; removing the identity-key guard therefore does not affect these verdicts.

7.3.4 Publishing the Network Identifier

To test that dependency, restore the guard on `longTermBPub` and change both declarations of `n` from **`knows private`** to **`knows public`**.

```
• SCUTTLEBUTT: RESULTS WITH PUBLIC N

Fail × confidentiality? n
Pass ✓ confidentiality? m1  [search exhausted at 2 sessions]
Fail × confidentiality? m2
Pass ✓ confidentiality? longtermapub  [search exhausted at 2 sessions]
Fail × authentication? Alice → Bob: secretbox1alice
Fail × authentication? Alice → Bob: secretbox2alice
Fail × authentication? Bob → Alice: secretbox1bob
Fail × authentication? Alice → Bob: secretboxm1alice
Fail × authentication? Bob → Alice: secretboxm2bob
Fail × equivalence? mastersecret2alice, mastersecret2bob

Fail × 8 of 10 queries failed.
```

With `n` public, every box loses authentication, Bob's reply `m2` loses confidentiality and the master secrets can be made to disagree. Removing the identity-key guard as well still does not change the verdicts. In this model, the additional failures follow from making the network identifier public, not from the identity-key guard. The traces for `secretBox2Alice` and `secretBox1Bob` carry the note that they are not minimized witnesses (§5.4); the other six are reproduced and grounded.

The `m2` trace does not require a long-term private key:

[2] A pre-shared key is secret material provisioned to participants before a session. It differs from public-key pre-authentication because all authorized peers know the same secret.

• SCUTTLEBUTT: HOW THE ATTACKER READS BOB'S REPLY

```
| 1. Attacker constructs PUBKEY(nil).
| 2. Attacker constructs MAC(n, PUBKEY(nil)).
| 3. Attacker replaces ephemeralapub, nmacalice (sent by
| Alice to Bob) with PUBKEY(nil), MAC(n, PUBKEY(nil)).
| (ephemeralapub was PUBKEY(ephemerala); nmacalice was
| MAC(n, ephemeralapub))
| 4. Attacker observes secretboxm2bob on the wire.
| 5. Attacker observes ephemeralbpub on the wire.
| 6. Attacker constructs ephemeralsecretbob.
| 7. Attacker observes longtermbpub on the wire.
| 8. Attacker constructs longtermsecretbob.
| 9. Attacker constructs mastersecret2bob.
| 10. Attacker observes n5 on the wire.
| 11. Attacker opens secretboxm2bob with mastersecret2bob
| and n5, obtaining m2.
> m2 (m2) is obtained by Attacker.
```

In step 3 the attacker replaces Alice's ephemeral public key and constructs a matching MAC with public `n`. Every key Bob then derives contains a Diffie–Hellman term whose other private value is `nil`, so the attacker reconstructs them in steps 6–9 from values it reads off the wire. Step 10 collects the nonce Bob sealed his reply under, which the attacker needs as well as the key, and step 11 opens the reply. No long-term private key appears anywhere in the trace, and no ciphertext is forged.

The traces for the two Alice-to-Bob boxes in this variant are where guard bypasses appear (§5.2.3). Bob's checked decryptions would fail under the mutated keys, but the attacker holds `masterSecret1Bob` and `masterSecret2Bob` and can therefore supply ciphertexts that each check accepts; the identity key Bob extracts from a bypassed box is attacker-known, so a signature under its private value `nil` verifies too.

Bob's signature check no longer authenticates Alice because the attacker controls the encryption context that delivers the signature. Guarding Bob's public key authenticates Bob to Alice; it does not authenticate Alice to Bob. With `n` public, `secretBox1Bob` joins the other boxes: the causal-order argument that protected it in one run does not survive a second one.

The queries for `m1` and `longTermAPub` still pass. These verdicts mean only that the search found no witness. An attack in Alice's direction may require a complete term that the protocol never computes, such as a forged transcript signature inside a newly encrypted box. When backward search needs such a term as a whole, it considers only protocol terms and terms already held by the attacker (§5.3). The passing verdicts must therefore be interpreted within that completeness limit.

7.3.5 Forward Secrecy

To test forward secrecy, return to the initial model with private `n` and guarded `longTermBPub`. Add a later phase that leaks Alice's long-term private key:

• SCUTTLEBUTT: ALICE'S LONG-TERM KEY IS COMPROMISED LATER

```
phase[1]

principal Alice[
    leaks longTermA
  ]
```

The attacker may use `longTermA` from phase 1 onward but cannot use it to alter phase 0 traffic. This distinguishes later compromise from a key known during the handshake.

• SCUTTLEBUTT: RESULTS AFTER LONG-TERM KEY COMPROMISE

```
Pass ✓ confidentiality? n  [search exhausted at 2 sessions]
Pass ✓ confidentiality? m1  [search exhausted at 2 sessions]
Pass ✓ confidentiality? m2  [search exhausted at 2 sessions]
Fail × confidentiality? longtermapub
Attack trace:
| 1. Attacker is handed longterma by a leaks declaration.
| 2. Attacker constructs longtermapub.
> longtermapub (longtermapub_bob) is obtained by Attacker.
Pass ✓ authentication? Alice → Bob: secretbox1alice  [search exhausted at 2
    sessions]
Pass ✓ authentication? Alice → Bob: secretbox2alice  [search exhausted at 2
    sessions]
Fail × authentication? Bob → Alice: secretbox1bob
Fail × authentication? Alice → Bob: secretboxm1alice
Fail × authentication? Bob → Alice: secretboxm2bob
Fail × equivalence? mastersecret2alice, mastersecret2bob

Fail × 5 of 10 queries failed.
```

Both message-confidentiality queries still pass. Only `longTermAPub` changes because the attacker can derive it from the leaked private key; the parenthetical in the conclusion names Bob's decrypted copy of that key, which resolves to the same value. The four failures of the base model were already present and are independent of the compromise. Under this model, the later identity-key leak does not reconstruct the ephemeral Diffie–Hellman terms used in the recorded session keys.

Moving the leak into phase 0 changes no verdict in this model. The attacker then holds Alice's key during the handshake, but the private network identifier still prevents it from producing a hello that Bob accepts, so it cannot use the key live. A later phase nevertheless remains the right construction for a forward-secrecy query, because it separates a later compromise from a key known during the handshake.

7.3.6 Findings

The four variants support four conclusions about this model:

- The initial confidentiality queries pass, and the message-confidentiality queries continue to pass after a later identity-key compromise.
- Reusing one key and empty associated data across transcript positions permits two type-confusion or reflection attacks within a single run, and a third once a second run can supply the replacement in time.
- The hello MAC binds an ephemeral key to a network but not to a session. Against a single run this only aborts the handshake; across two concurrent runs the attacker routes whole flights between them and the two parties complete on different master secrets.
- Making `n` public increases the failures from four to eight, whereas removing Bob's identity-key guard changes no verdict. The four additional failures therefore depend on secrecy of the network identifier.

CHAPTER 8

Post-Quantum Protocol Analysis

This chapter models a future adversary that records a protocol execution and later gains the ability to recover Diffie–Hellman private values. It then tests whether a hybrid post-quantum exchange protects the recorded message.

8.1 A Symbolic Quantum Threat Model

In a *harvest-now, decrypt-later* attack, an adversary records ciphertexts that are secure today and retains them until the underlying public-key problem becomes tractable. A sufficiently capable quantum computer running Shor's algorithm would break the discrete-logarithm assumptions used by Diffie–Hellman, allowing recorded session secrets to be reconstructed.

The symbolic model does not represent quantum computation or algorithmic hardness. It can represent the protocol-level consequence: every relevant Diffie–Hellman private value becomes available in a later phase.

• LEAKING EVERY DIFFIE--HELLMAN PRIVATE VALUE

```
phase[1]

principal Alice[leaks alongterm, ae1]
principal Bob[leaks blongterm, bs, bo]
```

In phase 0, the protocol runs under the ordinary active attacker. In phase 1, every listed Diffie–Hellman private value leaks (§2.6). The confidentiality query then asks whether a phase 0 message remains secret after that later disclosure.

The same capability can be expressed by weakening each affected public-key derivation:

• THE SAME ASSUMPTION, STATED DIRECTLY

```
galongterm = PUBKEY[weak from phase 1](alongterm)
gbs        = PUBKEY[weak from phase 1](bs)
```

The `PUBKEY[weak]` assumption lets the attacker recover a private value from the corresponding public key from phase 1 onward (§4.6). Ordinary reconstruction then derives every affected Diffie–Hellman secret.

The formulations differ in maintenance and scope. A `leaks` clause names particular private values and must be updated when the model adds a key. A weakening annotation states the failure at each public-key call site and automatically affects all uses of that term. Section 8.7 compares their traces.

Neither form models Shor's algorithm. Both are assumptions supplied by the model author about the information available to the future attacker.

The first analyses assume that the key-encapsulation mechanism remains secure while the classical exchange fails. It stands for ML-KEM [17] or another post-quantum KEM (§4.7.1). Verifpal cannot validate that computational assumption; it can only analyze its protocol consequences. A later variant reverses the assumption.

8.2 PQXDH

Signal's PQXDH construction [22] extends the X3DH handshake from Chapter 6. The model in this chapter uses all four X3DH Diffie–Hellman values derived from Bob's identity key, signed pre-key and one-time pre-key.

PQXDH adds a KEM encapsulation key signed by Bob's identity key. Alice encapsulates to it, obtaining shared secret `ss` and ciphertext `ct`. She includes `ss` with the four Diffie–Hellman values in the master secret and sends `ct` to Bob, who decapsulates it.

The resulting session key depends on both classical and post-quantum components. The intended hybrid property is that it remains protected while either component remains secure (§4.7.1).

8.3 A First Model

Bob signs both the Diffie–Hellman pre-key and the KEM encapsulation key with his identity key:

• PQXDH: BOB PUBLISHES HIS PRE-KEYS

```
principal Bob[
    knows public c1
    knows private blongterm, bs
    generates bo, dkb
    gblongterm = PUBKEY(blongterm)
    gbs = PUBKEY(bs)
    gbo = PUBKEY(bo)
    ekb = PUBKEY(dkb)
    gbssig = SIGN(blongterm, gbs)
    ekbsig = SIGN(blongterm, ekb)
  ]

Bob → Alice: [gblongterm], gbs, gbssig, gbo, ekb, ekbsig
```

Alice verifies both signatures, encapsulates to the KEM key and combines the KEM secret with four Diffie–Hellman values:

• PQXDH: ALICE INITIATES

```
principal Alice[
    generates m1, ae1, r
    gae1 = PUBKEY(ae1)
    valid = SIGNVERIF(gblongterm, gbs, gbssig)?
    kvalid = SIGNVERIF(gblongterm, ekb, ekbsig)?
    ss, ct = KEM_ENCAP(ekb, r)
    amaster = HASH(DH_KEX(gbs, alongterm), DH_KEX(gblongterm, ae1), DH_KEX(gbs,
        ae1), DH_KEX(gbo, ae1), ss)
    akenc = HKDF(amaster, c1, nil)
    generates n_e1
    e1 = AEAD_ENC(akenc, n_e1, m1,
      HASH(galongterm, gblongterm, gae1))
  ]

Alice → Bob: [galongterm], gae1, ct, n_e1, e1
```

Bob decapsulates the KEM ciphertext, reconstructs the master secret and checks Alice's ciphertext:

• PQXDH: BOB RESPONDS

```
principal Bob[
    bss = KEM_DECAP(dkb, ct)
    bmaster = HASH(DH_KEX(galongterm, bs), DH_KEX(gae1, blongterm), DH_KEX(gae1,
        bs), DH_KEX(gae1, bo), bss)
    bkenc = HKDF(bmaster, c1, nil)
    m1_d = AEAD_DEC(bkenc, n_e1, e1,
      HASH(galongterm, gblongterm, gae1))?
  ]
```

Both identity keys are guarded, so the model excludes identity-key substitution and isolates the future-compromise question. Alice checks both pre-key signatures. `KEM_DECAP` remains unchecked to model ML-KEM's implicit rejection (§4.5). The five-input `HASH` contains the four Diffie–Hellman values and the KEM secret.

Append the phase 1 leak clause from the start of the chapter. The queries test message confidentiality and Alice-to-Bob authentication:

• PQXDH: QUERIES

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

Because `dkb` remains secret, the intended hybrid argument predicts that the KEM component will preserve the message after every Diffie–Hellman private value leaks. The analysis exposes a key-role substitution that defeats this argument.

8.4 Analysis One: Missing Domain Separation

The confidentiality query is contradicted:

• PQXDH: FIRST MODEL

```
Fail × confidentiality? m1
Attack trace:
| 1. Attacker observes gbs on the wire.
| 2. Attacker observes gbssig on the wire.
| 3. Attacker replaces ekb, ekbsig (sent by Bob to Alice)
| with gbs, gbssig. (ekb was PUBKEY(dkb);
| ekbsig was SIGN(blongterm, ekb))
| 4. Alice's SIGNVERIF(gblongterm, gbs, gbssig)? passes —
| the attacker controls one of its inputs.
| 5. Attacker observes e1 on the wire.
| 6. Attacker is handed alongterm by a leaks declaration.
| 7. Attacker constructs DH_KEX(gbs, alongterm).
| 8. Attacker observes gblongterm on the wire.
| 9. Attacker is handed ae1 by a leaks declaration.
| 10. Attacker constructs DH_KEX(gblongterm, ae1).
| 11. Attacker constructs DH_KEX(gbs, ae1).
| 12. Attacker observes gbo on the wire.
| 13. Attacker constructs DH_KEX(gbo, ae1).
| 14. Attacker observes ct on the wire.
| 15. Attacker is handed bs by a leaks declaration.
| 16. Attacker opens ct with bs, obtaining ss.
| 17. Attacker constructs amaster.
| 18. Attacker constructs akenc.
| 19. Attacker observes n_e1 on the wire.
| 20. Attacker opens e1 with akenc and n_e1, obtaining m1.
> m1 (m1) is obtained by Attacker.
Pass ✓ authentication? Alice → Bob: e1  [search exhausted at 2 sessions]

Fail × 1 of 2 queries failed.
```

Steps 14–16 show the critical dependency. The attacker decapsulates Alice's KEM ciphertext with `bs`, the private value for Bob's Diffie–Hellman signed pre-key, after `bs` leaks in phase 1. Step 4 is the gate at which Alice's signature check accepts the substituted key. Alice had computed:

```
KEM_ENCAP(PUBKEY(bs), r)
```

In step 3, the attacker replaces the KEM key `ekb` with the Diffie–Hellman pre-key `gbs`, and replaces its signature with the valid `gbssig`. Both values came from Bob's public

bundle. Alice's signature check succeeds because Bob did sign `gbs`; neither the signed message nor the verification context identifies the key's intended role.

Alice therefore encapsulates to a key whose private value belongs to the classical exchange. The supposed post-quantum component becomes another classical component and fails under the same future disclosure.

Bind Signatures to Key Roles

A signature authenticates its message, but the message must also encode the value's protocol role. If two key types are signed under the same identity key without distinct tags, a valid signed value from one role may be accepted in the other. The signature is valid; its context is wrong.

The phase 0 substitution uses only public values already in Bob's bundle. The phase 1 compromise reveals the private value that completes the attack; it does not create the type-confusion flaw.

8.5 Analysis Two: Domain Separation

Domain separation binds each signature to a key role. Assign a public tag to each pre-key type and sign the tagged hash:

• PQXDH: DOMAIN-SEPARATED PRE-KEY SIGNATURES

```
principal Bob[
    knows public c1, c2, c3
    ...
    gbssig = SIGN(blongterm, HASH(c2, gbs))
    ekbsig = SIGN(blongterm, HASH(c3, ekb))
  ]

principal Alice[
    valid = SIGNVERIF(gblongterm, HASH(c2, gbs), gbssig)?
    kvalid = SIGNVERIF(gblongterm, HASH(c3, ekb), ekbsig)?
    ...
  ]
```

A signature over `HASH(c2, gbs)` cannot satisfy a verification over `HASH(c3, ekb)`. The cross-role substitution now fails:

• PQXDH: WITH DOMAIN SEPARATION

```
Pass ✓ confidentiality? m1  [search exhausted at 2 sessions]
Pass ✓ authentication? Alice→ Bob: e1  [search exhausted at 2 sessions]

Pass ✓ All 2 queries pass.
```

After the phase 1 leaks, the attacker reconstructs all four Diffie–Hellman components but not `ss`. Verifpal finds no contradiction to message confidentiality or authentication under these assumptions.

Guarding `ekb` also prevents the substitution, but represents a different mechanism. Domain separation makes the signature distinguish key roles. A guard assumes that Alice authenticated the exact KEM key before the exchange. Either assumption blocks this trace, but only the first is provided by the signed bundle itself.

8.6 Analysis Three: Classical X3DH Under Future Compromise

To isolate the KEM's contribution, remove the KEM keys, signatures, encapsulation and decapsulation. The resulting classical X3DH master secret contains only the four Diffie–Hellman values. Keep the phase 1 leak unchanged.

• CLASSICAL X3DH: HARVEST NOW, DECRYPT LATER

```
Fail × confidentiality? m1
Attack trace:
| 1. Attacker observes e1 on the wire.
| 2. Attacker observes gbs on the wire.
| 3. Attacker is handed alongterm by a leaks declaration.
| 4. Attacker constructs DH_KEX(gbs, alongterm).
| 5. Attacker observes gblongterm on the wire.
| 6. Attacker is handed ae1 by a leaks declaration.
| 7. Attacker constructs DH_KEX(gblongterm, ae1).
| 8. Attacker constructs DH_KEX(gbs, ae1).
| 9. Attacker observes gbo on the wire.
| 10. Attacker constructs DH_KEX(gbo, ae1).
| 11. Attacker constructs amaster.
| 12. Attacker constructs akenc.
| 13. Attacker observes n_e1 on the wire.
| 14. Attacker opens e1 with akenc and n_e1, obtaining m1.
> m1 (m1) is obtained by Attacker.
Pass ✓ authentication? Alice→ Bob: e1  [search exhausted at 2 sessions]

Fail × 1 of 2 queries failed.
```

The confidentiality trace contains no mutation. The attacker observes the public values and, after the later leaks, reconstructs the four Diffie–Hellman secrets and derives Alice's message key. The minimized trace shows only the leaked private values needed for that derivation.

This is the harvest-now, decrypt-later threat: passive recording in phase 0 followed by key recovery in phase 1. Unlike a substitution or reflection attack, it creates no anomalous phase 0 message for a participant to detect.

Authentication still passes in the classical model. The attacker derives the session key only in phase 1, whereas `e1` was sent in phase 0. A substitution is judged against the knowledge available in the earliest phase in which the value is communicated (§2.6), so the later leaks cannot justify a forged ciphertext in the earlier phase. The recorded message can be read after the compromise, but it could not have been forged before it. A leak placed in phase 0 would permit the forgery as well, and would model a key known during the handshake rather than a later compromise.

8.7 Modeling Failure with Weakening Assumptions

The preceding analyses enumerate five phase 1 leaks. An alternative is to remove those leaks and annotate every Diffie–Hellman public-key derivation in the domain-separated model. Leave the KEM key `ekb` unannotated so that it remains the secure hybrid component:

• PQXDH: THE WEAKENING ASSUMPTION FORMULATION

```
principal Bob[
    ...
    gblongterm = PUBKEY[weak from phase 1](blongterm)
    gbs        = PUBKEY[weak from phase 1](bs)
    gbo        = PUBKEY[weak from phase 1](bo)
    ekb        = PUBKEY(dkb)
  ]

phase[1]
```

This form produces the same passing verdicts as Analysis Two, printed beneath the same five warnings. Removing the KEM again produces the classical failure, with a trace that attributes private-key recovery to the declared assumption:

• CLASSICAL X3DH, UNDER A DECLARED ASSUMPTION

```
Warning ▲ Analysis performed under 5 declared weakening
assumptions:
Warning ▲ PUBKEY[weak from phase 1](alongterm)
Warning ▲ PUBKEY[weak from phase 1](blongterm)
Warning ▲ PUBKEY[weak from phase 1](bs)
Warning ▲ PUBKEY[weak from phase 1](bo)
Warning ▲ PUBKEY[weak from phase 1](ae1)

Fail × confidentiality? m1
Attack trace:
| 1. Attacker observes e1 on the wire.
| 2. Attacker observes gbs on the wire.
| 3. Attacker observes galongterm on the wire.
| 4. Attacker breaks galongterm under the declared `weak'
| assumption, obtaining alongterm.
| 5. Attacker constructs DH_KEX(gbs, alongterm).
| 6. Attacker observes gblongterm on the wire.
| 7. Attacker observes gae1 on the wire.
| 8. Attacker breaks gae1 under the declared `weak'
| assumption, obtaining ae1.
| 9. Attacker constructs DH_KEX(gblongterm, ae1).
| 10. Attacker constructs DH_KEX(gbs, ae1).
| 11. Attacker observes gbo on the wire.
| 12. Attacker constructs DH_KEX(gbo, ae1).
| 13. Attacker constructs amaster.
| 14. Attacker constructs akenc.
| 15. Attacker observes n_e1 on the wire.
| 16. Attacker opens e1 with akenc and n_e1, obtaining m1.
> m1 (m1) is obtained by Attacker.
Pass ✓ authentication? Alice→ Bob: e1  [search exhausted at 2 sessions]

Fail × 1 of 2 queries failed.
```

Verifpal lists the five annotations once each, although session replication applies them to every session's copy of the annotated terms (§5.6.3). In Analysis Three, a **leaks** declaration hands `alongterm` to the attacker. Here, steps 4 and 8 derive `alongterm` and `ae1` by breaking the corresponding public keys, and each step names the assumption it depends on. The resulting attacker knowledge is the same, but the annotation states the assumed cryptographic failure directly. The authentication query passes for the same reason as before: the assumption is in force only from phase 1.

Verifpal prints active assumptions before every result, including passing results. The Analysis Two verdict is therefore explicitly conditional on the KEM remaining secure after the declared classical break.

8.8 Analysis Four: Post-Quantum Component Compromise

The converse experiment compromises the KEM while leaving Diffie–Hellman intact. In the domain-separated model, replace the classical leaks with a phase 1 leak of `dkb`:

• PQXDH: WITH THE KEM COMPROMISED INSTEAD

```
Pass ✓ confidentiality? m1  [search exhausted at 2 sessions]
Pass ✓ authentication? Alice → Bob: e1  [search exhausted at 2 sessions]

Pass ✓ All 2 queries pass.
```

Verifpal again finds no contradiction because the four Diffie–Hellman values remain unavailable. This is the intended hybrid guarantee: either uncompromised component is sufficient to protect the derived key.

If phase 1 reveals both the Diffie–Hellman private values and `dkb`, the confidentiality query fails: the trace decapsulates `ct` with the leaked `dkb` and then follows the classical derivation. The hybrid property ends when neither component remains secret. The authentication query still passes, because the compromise occurs after the message was sent.

8.9 Findings

The analyses support three conclusions:

- A correctly composed hybrid exchange protects the message when either the classical or post-quantum component remains secure.
- If both components fail in this model, the attacker derives the session key and reads the recorded message. The authentication query still passes, because the compromise occurs after the message was sent and Verifpal does not let later knowledge justify an earlier forgery.
- Signing two key types without domain separation permits a valid signed Diffie–Hellman key to replace the KEM key. Guards, checked verification, fresh KEM randomness and hybrid derivation do not prevent this type-confusion attack.

The third result arises from a missing binding between a key and its protocol role. The security of the signature primitive alone does not rule out that substitution.

Bibliography

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

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

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

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

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

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

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

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

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

[10] Nadim Kobeissi. The Verifpal analysis engine: Semantics, validation, and evaluation. Preprint, Symbolic Software, 2026. https://github.com/symbolicsoft/verifpal-papers.

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

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

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

[14] 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.

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

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

[17] National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard. Technical Report FIPS 203, U.S. Department of Commerce, 2024.

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

[19] Deirdre Connolly, Chelsea Komlo, Ian Goldberg, and Christopher A. Wood. The Flexible Round-Optimized Schnorr Threshold (FROST) Protocol for Two-Round Schnorr Signatures. RFC 9591, June 2024.

[20] 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.

[21] 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.

[22] Ehren Kret and Rolfe Schmidt. The pqxdh key agreement protocol. Technical report, Signal Messenger, 2023.

Reference Figures

model ::= *attacker principal* (*principal* | *message* | *phase*)+ *scenarios*? *queries*

attacker ::= `attacker[` (`active` | `passive`) `]`

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

knows ::= `knows` (`private` | `public`) *constant* (`,` *constant*)*

generates ::= `generates` *constant* (`,` *constant*)*

leaks ::= `leaks` *constant* (`,` *constant*)*

assignment ::= *constant* (`,` *constant*)* `=` *primitive*

message ::= *string* (→ | `->`) *string* `:` (*constant* | *guardedConstant*) (`,` (*constant* | *guardedConstant*))*

phase ::= `phase[` *number* `]`

scenarios ::= `scenarios[` *scenario*+ `]`

scenario ::= *string* `[` *constant* `=` *constant* (`,` *constant* `=` *constant*)* `]`

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* *annotation*? `(` (*constant* | *primitive*) (`,` (*constant* | *primitive*))* `)` [`?`]

annotation ::= `[` *annotationItem* (`,` *annotationItem*)* `]`

annotationItem ::= *threshold* | *capability*

threshold ::= *number*

capability ::= (`weak` | `forgeable` | `malleable`) (`from phase` *number*)?

primitiveName ::= `BLIND` | `UNBLIND` | `RINGSIGN` | `RINGSIGNVERIF` | `HASH` | `HKDF` | `AEAD_ENC` | `AEAD_DEC` | `ENC` | `DEC` | `MAC` | `ASSERT` | `CONCAT` | `SPLIT` | `SIGN` | `SIGNVERIF` | `PKE_ENC` | `PKE_DEC` | `THRESHOLD_SPLIT` | `THRESHOLD_JOIN` | `THRESHOLD_SIGN` | `PUBKEY` | `DH_KEX` | `KEM_ENCAP` | `KEM_DECAP`

FIGURE 1 Verifpal language syntax.

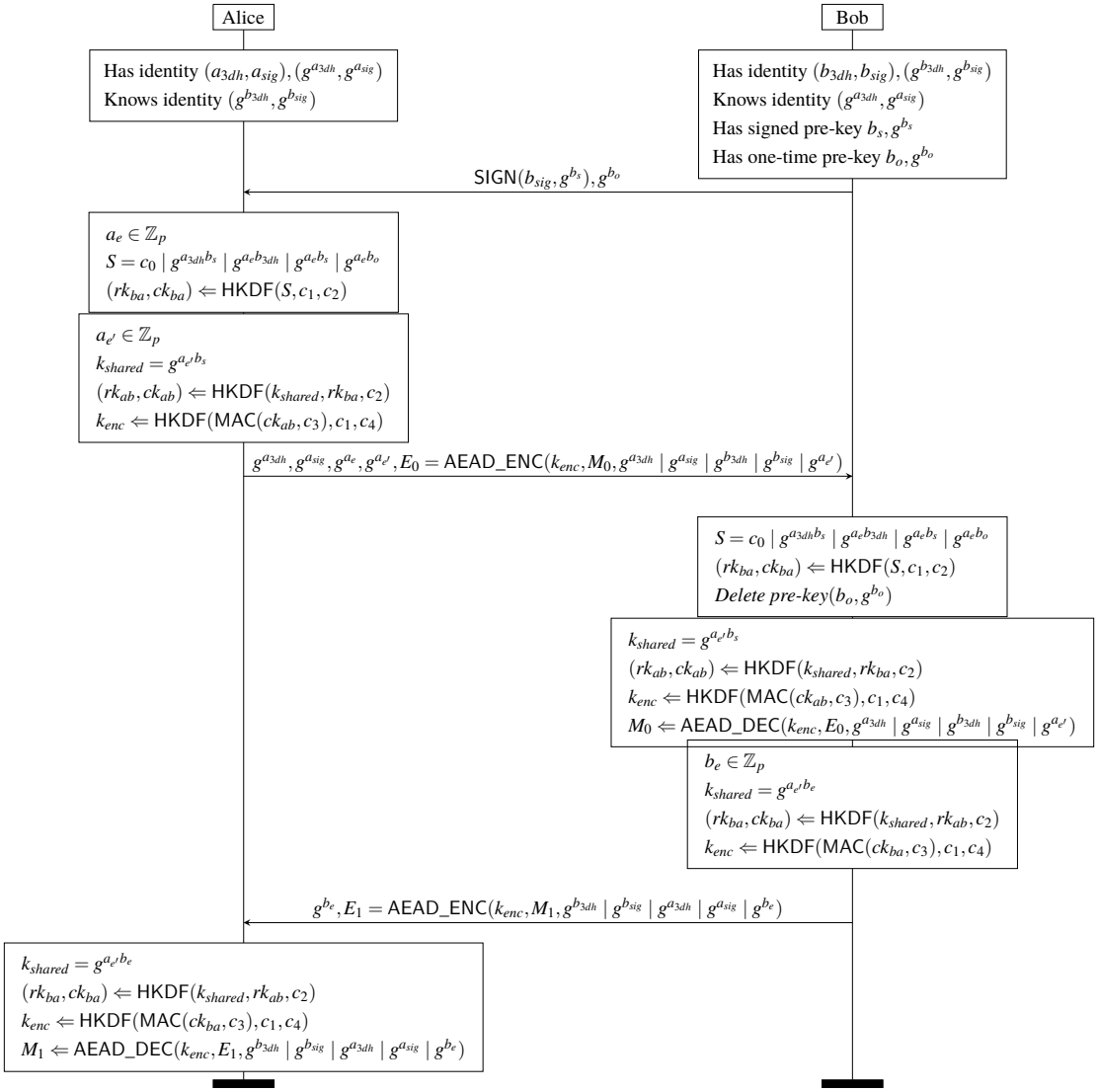


FIGURE 2 A simplified Signal exchange. Alice obtains Bob's signed pre-key through the server, completes her side of the key exchange and sends the initial message M_0. Bob then derives the same session state and decrypts M_0.

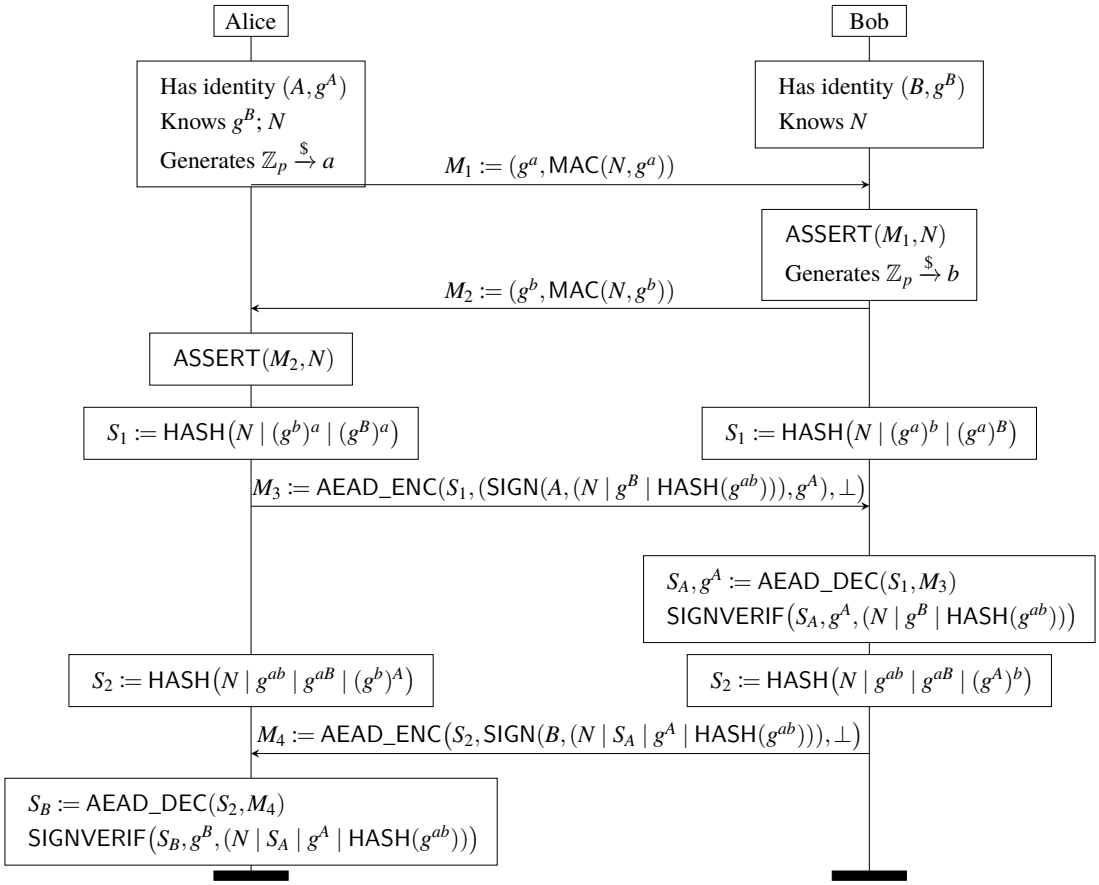


FIGURE 3 Secure Scuttlebutt's authenticated key exchange. Bob acts as the server, and Alice begins with an authenticated copy of Bob's long-term public key g^B. The exchange aims to hide Alice's identity and provide key-compromise impersonation resistance and forward secrecy.