IN2194
Peer-to-Peer Systems and Security
Summer Semester 2023

Final Report

Team Gossiphers (Gossip-1):
Yannis Matezki and Dominik Ritzenhoff

Date Submitted: Sep. 05, 2023

Changes to Midterm Assumption

- Added timestamp to header to mitigate replay attacks
- Each packet is prefaced with an RSA-Encrypted AES key and IV that is used for the encryption of the packet itself
- Signature size changed from 64 bytes to 512 bytes, which matches the size of the RSA key (4096 bits).

Architecture of the Module

Logical Structure

Go does not offer classes, but structs with method implementations offer a similar functionality.
Our module is separated into a few logical units:

- API
    - TCP Server that answers API requests defined in the given specification
    - API Packet serialization/deserialization
    - Basic logic for API clients to listen to events of a certain type
    - Structs used like classes:
        - *Server*
            - Stateful TCP API server, stores connections registered for events
- Config
    - Configuration values are read from a .ini file, which serve as adjustable parameters within the program.
- Gossip
    - UDP Server and Client that answer and initiate P2P push and pull requests as well as regular Ping probes defined within the Brahms algorithm, respectively.
    - P2P Packet serialization/deserialization.
    - Structs used as classes:
        - *View* - Represents a view within Brahms algorithm.
        - *Sampler* - Represents a sampler within Brahms algorithm.
        - *SamplerGroup* - Represents a group of samples belonging to one node.
        - *Node* - Represents a node (i.e. peer) within the network.
        - *Gossip* - Represents the collection of data structures needed to make the Brahms algorithm function.
        - *Server* - Stateful UDP P2P API server. Stores connections registered for events.
        - *Identity* - Represents the SHA256 hash of the RSA public key.
        - *Server* - Represents a UDP listener with handlers for gossip-related messages.
        - *Crypto* - Represents a container for all the cryptographic functionality within the gossip protocol.
- Challenge
    - Module to generate computational challenges in the required format and check the validity of the produced solutions
    - Utilized a ring of rotating keys to invalidate old challenges after a given timeframe without storing each generated challenge
    - Structs used like classes:
        - *Challenger*
            - Stores internal key ring and automatically rotates keys in a given time frame
            - Generates challenges and validates solutions using the internal keys

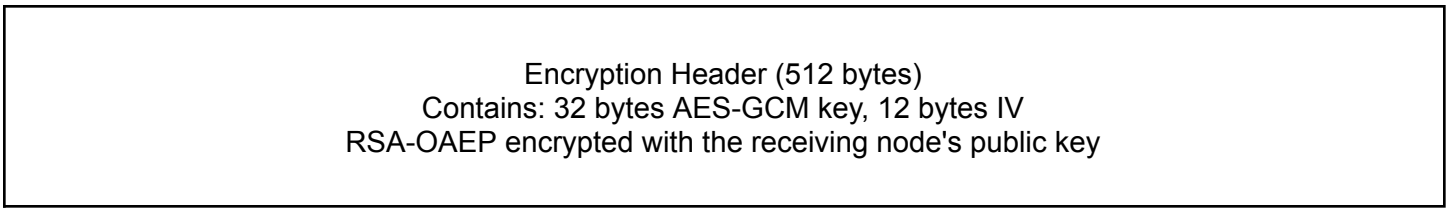
Process Architecture

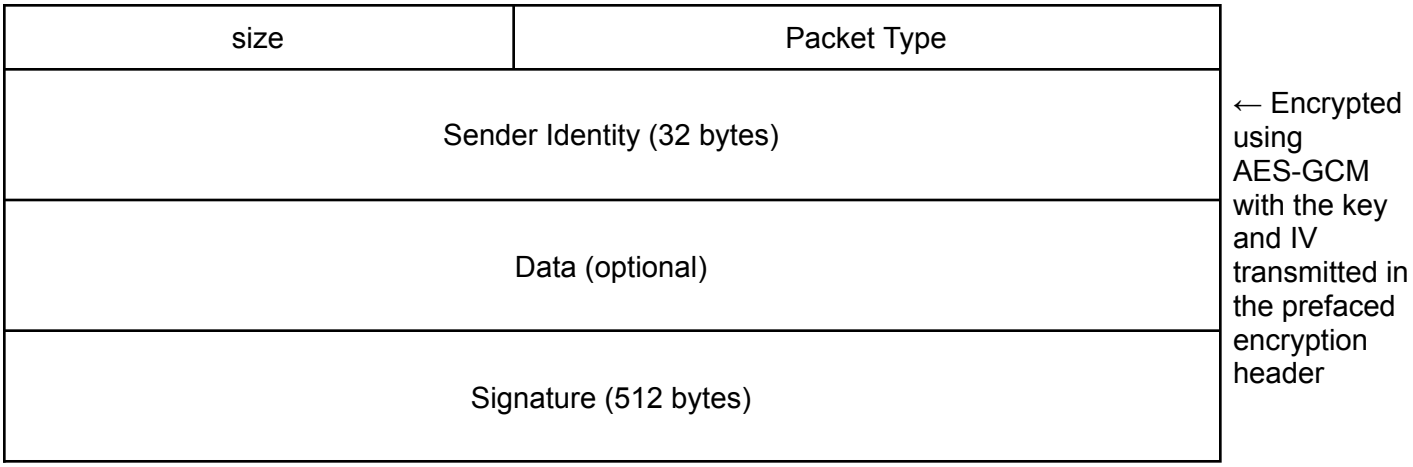
Goroutines are utilized for all internal processes, acting as lightweight threads managed by the Go runtime. Each TCP/UDP server is running in its own Goroutine and spawns a new Goroutine for each incoming connection/ packet to allow for parallel processing of the received data. Goroutines are also utilized for periodically executed tasks, like the key rotation of the challenge generator.
Shared data is guarded by Mutex locks, in cases with many possible writing threads RWMutex is utilized.

Networking

Packets that are not formatted correctly are simply dropped, a flooding of well formatted packets is currently not prevented, but the local view should not be impacted due to other security measures.
Communication between peers is UDP-based and assumes the network to be unreliable.
Each peer locally tracks the state of communication flows with other peers and which packets to expect within a certain time frame. Packets that are not expected (i.e. unsolicited Pull Responses) are simply ignored.

General Structure:





Gossip Ping:

- Sent to probe another node. Another node should reply with a GOSSIP PONG if available.
- Packet Type: GOSSIP PING = 0x00 0x30
- No data

Gossip Pong:

- A response to the ping, indicating that this node is still alive.
- Packet Type: GOSSIP PONG = 0x00 0x31
- No data

Gossip Pull Request:

- Sent to request a list of nodes from another node, the other node should respond with a GOSSIP PULL RESPONSE
- Packet Type: GOSSIP PULL REQUEST = 0x00 0x40
- No data

Gossip Pull Response:

- Response to the pull request with the node's view included in the body.
- Packet Type: GOSSIP PULL RESPONSE = 0x00 0x41

nodes (List of peers, Identity and Address separated by tab (\t), peers separated by newline (\n)) → <Identity>\t<Address>\n

Gossip Push Request:

- Request a challenge to push own address to another node, other node should respond with a GOSSIP PUSH CHALLENGE to this
- Packet Type: GOSSIP PUSH REQUEST = 0x00 0x50
- No Data

Gossip Push Challenge:

- Response to the push request. Includes a challenge token for the destination node to 'solve' in order to have its gossip push request acknowledged.
- Packet Type: GOSSIP PUSH CHALLENGE = 0x00 0x51

Difficulty
Challenge (32 bytes)

Gossip Push:

- After solving the challenge received in a GOSSIP PUSH CHALLENGE packet, a node can then continue to send its own ID/address to another node with the nonce that acts as the solution to the challenge
- Packet Type: GOSSIP PUSH = 0x00 0x52

Challenge (32 bytes)
Nonce (8 bytes)
<Identity>\t<address>\n

Of own peer

Gossip Message:

- Contains a gossip message that should be spread to all nodes in the local view when received from a known peer
- TTL should be decreased every time the message is forwarded, a message with a TTL of 1 is not forwarded further
- Packet Type: GOSSIP MESSAGE = 0x00 0x60

TTL	reserved	data type
data		

Security Measures

Every message exchanged between peers contains the sender's identity and is signed by the sender using its public key. Additionally, it is encrypted using the technique specified above. The receiver discards any packet that it cannot decrypt and consecutively verify the signature of. An attacker can still guess the packet type based on its size, but both integrity and confidentiality of the contents is guaranteed.
As seen in the data blocks above, we also include a gossip push challenge to make it more difficult for a group of malicious peers to push-flood another peer using i.e. a sybil attack.
This challenge is generated using the other peers' identity.
To significantly lower the impact of replay attacks, all packets that carry a timestamp older than 8 seconds are also discarded.
Additionally only one push from each identity is allowed in one Brahms Gossip cycle.
This limits the amount of pushes an attacker can send with respect to available compute and network addresses.
Finally the parsing is made to expect malformatted messages and deal with them safely by only receiving and parsing an expected, limited amount of bytes.

Building/ Running the Software

Prerequisites

Local Build and Testing:

- Go 1.20+
- Docker

Running a prebuilt Image:

- Docker

Building/ Testing

For local development, first download the required dependencies by executing the following command in the projects root directory

- `*go mod download*`

To run unit tests, `cd` into the project's root folder and issue the following commands:

- `*go test ./…*` → this runs all the tests existing within the Go project.

To build an executable, `cd` into the project's root folder and issue the following commands:

- `*cd cmd/gossip/*` → this brings you to the `main.go` file, which is typically where execution begins for go projects.
- `*go build .*` → this builds the project and outputs the binary called 'gossip' in your CWD.
- `*./gossip -c <configuration file path>*` → this runs the executable. Defaults to `$CWD/config.ini` if you don't add the -c flag.

The project also comes with a small go command-line tool to quickly generate configurations and keys for an arbitrary amount of nodes and spin up networked docker containers for an end-to-end test. The tool requires the docker API to be configured and accessible by the running user.
The tool resides in the e2e-tests directory and can be used by executing the following commands in the terminal:

- `*go run ./main.go start -n [number of nodes]*` → Start the given number of nodes in separate docker containers, connect them with a bridge network and bind the first nodes API port to the local port 7001
- `*go run ./main.go stop*` → Stop all running nodes, delete the containers and clean up the generated data and the docker network

Logs for each container can be acquired through the regular docker cli commands.

Running

For local builds, refer to the steps in the previous section.
Docker builds can be run by either building the image yourself with the provided Dockerfile, or pulling it from the GitLab container registry (`*netintum/teaching/p2psec_projects_2023/gossip-1*`).
Simply bind the configuration file to the path `/config.ini` in the container and publish the required ports according to your configuration. Additionally, the hostkey and peer keys will also have to be bound to the respective configured filepaths.
Example command:
`*docker run -v config.ini:/config.ini -v hostkey.pem:/hostkey.pem -v ./peerkeys/:/peerkeys/ -p 7001:7001 -p 7002:7002/udp [image name]*`

Configuration

The configuration is given in an .ini file, the file path of which can be passed to the executable at launch.
It can contain the following configuration keys:

Key within the Gossip section	**Required**	**Default**	**Description**
degree	false	30	Internal view size of the gossip module. This is the number of nodes that we could directly communicate with within a single gossip round.
l2	false	30	Size of the SamplerGroup for history samples.
weight_push weight_pull weight_history	false	45, 45, 10	Weights of the different sources for nodes when recalculating the local view. Should add up to 100.
bootstrap_nodes	false		BootstrapNodesStr is a list of node components in the following form → nodes = <addr1>,<id1>|<addr2>,<id2>|...|<addrn>,<idn>|
rounds_between_pings	false	8	Number of gossip rounds before each node in the sampler is probed with a Ping packet.
api_address	false	"localhost:7001"	The address the API server will bind to, which adheres to the API specification given for Gossip.
gossip_address	false	“localhost:7002”	The address the gossip server will bind to, this address will also be shared, so it should be reachable by other peers.
hostkeys_path	false	./hostkeys/	Path to the folder which contains the public keys for all other peers. The keys are expected to have their peers' hex-encoded identity as their filename.
challenge_difficulty	false	19	Represents the difficulty in solving a computational puzzle. The greater the difficulty, the more time it will take to solve the puzzle.
challenge_max_solve_ms	false	300	Represents the max amount of time a peer is given to solve the puzzle.

Additionally there’s the `hostkey` configuration key in the root section of the .ini file, which configures the path for the peers private key .pem file. This configuration key is required.

Future Work

Currently, each packet carries its own AES key and signature, which is very inefficient. Instead of this, peers could negotiate an expiring AES encryption key every so often, which is RSA encrypted and signed to achieve the same level of security with a much smaller overhead. Since the AES key is only known to this one identity, which has signed it when negotiating it, any packet encrypted using that key could be considered signed.

Another improvement could be integrating the Network Size Estimator with the Gossip module, which would create a more dynamic system.

Workload Distribution

Together:

- We collectively wrote the reports, designed the protocol and came up with a design for the gossip implementation specification.
- Reviewing each other's work in the form of Merge Requests

Yannis:

- API Server (Packet parsing, Client registration, etc.)
- Small Config module providing parsed configuration values and defaults
- Brahms Sampler and SamplerGroup for historical view samples
- Challenge module for generation, solving and validation of PoW challenges
- Gossip UDP Server (handlers for each kind of packet, calling other modules for encryption/ decryption, challenge generation/ solving and distribution of messages)
- E2E testing tool which starts a number of nodes and cleans up afterwards
- GitLab CI/CD setup for compilation and docker image creation
- Implementation of packet timestamping against replay attacks
- Final E2E testing and fixing of the found issues

Dominik:

- Strung together Gossip components and implemented internal View management.
- Implemented encryption, decryption, packet signing and signature verification.
- Node and Identity
- Parsing and serializing Gossip packets.
- Parsing and Writing (i.e. ToBytes) for Gossip packets
- Lots of refactoring as components were added and removed.
- Tests for all the aforementioned topics
- Documentation

Effort spent for the Project

In total, we spent a combined amount of roughly 20-24 working days on the project, out of which approximately 10-12 fell on each of us. As we mentioned in the midterm report, we distributed the workload in such a way as to make the time investment for both parties more even.