

Représentation et analyse du réseau de confiance OpenPGP

Antoine Amarilli

Table des matières

1	Position du problème	2
1.1	Cryptographie asymétrique	2
1.2	Réseau de confiance	2
1.3	OpenPGP	2
2	Étude du réseau de confiance	2
2.1	Objectifs du TIPE	2
2.2	État actuel de la recherche	3
3	Analyse du graphe	3
3.1	Importation	3
3.2	Parcours	3
3.3	Lien entre TLD et distances	3
4	Représentation du graphe	4
4.1	Critères esthétiques	4
4.2	Choix d'un algorithme	4
4.3	Algorithme force-directed	4
4.4	Interface	5
4.5	Observations	5
5	Prolongements envisageables	5

1 Position du problème

1.1 Cryptographie asymétrique

La **cryptographie** s'attache à la protection de la confidentialité, de l'intégrité et de l'authenticité des messages. La **cryptographie asymétrique** procède en associant à chaque utilisateur une **clé privée** gardée secrète et une **clé publique** diffusée à ses correspondants. La clé publique permet de chiffrer des messages et de vérifier des signatures ; la clé privée permet de déchiffrer des messages et d'apposer des signatures (voir figure 1 p. 8).

Cependant, une attaque de l'homme du milieu peut être menée lors de l'échange des clés publiques (voir figure 2 p. 9). Cela rend nécessaire le recours à un canal sans risque d'attaque active.

1.2 Réseau de confiance

Plusieurs solutions permettent de pallier ce problème : avoir recours à une autorité de certification centrale, ou utiliser le **réseau de confiance**, que nous étudierons ici. Son fonctionnement est schématisé par la figure 3 p. 13. Remarquons qu'il n'offre pas de transitivité (voir figure 4 p. 13).

En pratique, les correspondants du cryptosystème conservent, en plus de leur clé publique, toutes les signatures apposées par des tiers, et échangent leurs clés par l'intermédiaire de serveurs de clés se synchronisant les uns aux autres.

Lorsque deux personnes désirent signer leurs clés, elles se rencontrent physiquement, offrent chacune une garantie de leur identité (en général une carte d'identité ou un passeport), échangent les empreintes de leurs clés par une fonction de hachage cryptographique pour s'assurer que les clés n'ont pas été falsifiées pendant le transfert, et vérifient que les adresses de courriel sont correctes. Certaines associations organisent des fêtes de signature de clé, où de nombreux participants se rencontrent simultanément pour signer leurs clés.

L'ensemble des clés et des signatures de clés forme le réseau de confiance. Il s'interprète naturellement comme un graphe orienté ayant pour sommets les clés de chiffrement et comme arêtes les signatures d'une clé par une autre.

1.3 OpenPGP

OpenPGP est un standard de cryptographie asymétrique couramment utilisé actuellement, décrit par [13]. Les logiciels Pretty Good Privacy (PGP) et GNU Privacy Guard (GPG) en sont des implémentations compatibles, la seconde étant libre et gratuite.

Le site [6] propose une version téléchargeable et régulièrement mise à jour à partir des serveurs de clés¹ de la plus grande composante fortement connexe du réseau de confiance : il fournit la liste des signatures entre clés, ainsi que le nom et l'adresse de courriel indiqués par le créateur de chaque clé.

À l'heure actuelle, le plus grand ensemble fortement connexe comporte un peu plus de 40000 clés, chacune étant signée en moyenne par environ 10 autres clés (soit environ 400000 signatures).

2 Étude du réseau de confiance

2.1 Objectifs du TIPE

Mon TIPE vise à concevoir un programme permettant de mener des analyses sur le réseau de confiance OpenPGP et d'en fournir une représentation exploitable. En particulier, j'ai cherché à mettre en relation la position d'une clé dans le réseau de confiance et les informations géographiques que permet de déterminer le TLD (*Top Level Domain*) de l'adresse de courriel qui lui est associée.

Une telle étude présente plusieurs applications potentielles. Une corrélation forte entre proximité géographique des clés et proximité dans le graphe permettrait d'inférer avec précision l'origine géographique d'une clé quelconque. Enfin, comme les signatures de clés nécessitent généralement une rencontre physique de leurs propriétaires, une identification de traits caractéristiques du réseau de confiance provenant de cette réalité humaine pourrait permettre de le distinguer de réseaux générés artificiellement : cela offrirait une protection contre un attaquant qui réaliserait un double factice des clés du réseau de confiance, et rendrait possible la détection d'éventuels sous-réseaux suspects dans le graphe.

¹La synchronisation n'est cependant pas parfaite ; le serveur à partir duquel [6] extrait les informations nécessaires n'est pas systématiquement à jour.

2.2 État actuel de la recherche

Diverses études informelles du réseau de confiance OpenPGP ont été déjà menées.

Le site [6] analyse un motif en forme de feuille dans des représentations du réseau de confiance où les clés sont classées par leur distance moyenne aux autres clés, placées sur deux axes orthogonaux, et où un point blanc est positionné à l'intersection des lignes et des colonnes correspondant à des clés qui se sont signées. Différentes variations de cette représentation y sont étudiées (restriction à un TLD, classement par TLD, chemins de longueur 2, autres critères de tri), avec application à un réseau de confiance aléatoire. Le site propose également le logiciel wotsap, qui permet de calculer des statistiques générales sur le réseau de confiance, des statistiques pour une clé, des chemins entre couples de clés, des représentations graphiques, et une liste des signatures qui seraient les plus utiles au réseau de confiance. Enfin, il fournit des exports réguliers du réseau de confiance, que j'ai utilisés.

Le site [7] propose divers outils liés au réseau de confiance : recherche de chemins entre deux clés, classement des clés par distance moyenne aux autres clés, évolution des statistiques pour les clés individuelles, évolution temporelle du nombre de clés, du nombre de signatures par clés et de la distance moyenne entre clés, distribution du degré des sommets et des distances, comportement du graphe lors de la suppression de sommets aléatoires. Certaines de ces statistiques sont générées avec wotsap, mais les données sont extraites d'autres sources.

Des analyses plus anciennes sont proposées par [8], [9] et [10].

Pour ce qui est du dessin du réseau de confiance, le logiciel sig2dot [11] permet de convertir des trousseaux de clés OpenPGP en des graphes qui peuvent être fournis à des logiciels généralistes de dessin de graphe. Cependant, le format standard des trousseaux de clés se prête mal à l'importation de l'intégralité du réseau de confiance, et peu de logiciels sont en mesure de représenter un graphe aussi grand en un temps raisonnable.

3 Analyse du graphe

3.1 Importation

Le graphe du réseau de confiance est importé à partir des données fournies par [6] sous forme d'un fichier Wotsap (voir [14] pour la spécification). Le fichier représentant le graphe de confiance actuel fait environ 1,5 mégaoctets (le format Wotsap vise à être aussi compact que possible).

Le langage choisi pour la rédaction du programme est C, en raison de la vitesse d'exécution que cela permet d'atteindre, ce qui est nécessaire au vu de la taille du graphe. Le code source complet du programme rédigé pour le TIPE est donné en annexe.

Le graphe est représenté sous la forme de listes d'adjacence indiquant, pour chaque sommet, la liste des arêtes qui en partent et qui y arrivent. Ce choix est motivé par la faible densité du graphe, qui rend cette représentation préférable aux matrices d'adjacence (voir [4], p. 503).

3.2 Parcours

J'ai implémenté l'algorithme de parcours du graphe en largeur d'abord, tel que décrit par [5]. Il permet de calculer la distance des clés du graphe à une clé arbitraire, et donc, en faisant cela pour toutes les clés, la distance moyenne entre les couples de clés. Ce résultat est déjà calculé par wotsap.

3.3 Lien entre TLD et distances

Puisque la signature de clés nécessite une rencontre physique entre signataires, que de telles rencontres ont le plus souvent lieu entre habitants d'un même pays, et que le TLD des adresses de courriel correspond parfois à un pays, on peut s'attendre à ce que la distance moyenne entre deux clés d'un même TLD soit plus faible que celle entre deux clés aléatoires (si ce TLD correspond à un pays).

Pour vérifier cette conjecture, le programme calcule la somme des distances entre tous les couples de clés d'un même TLD, et fait la même chose pour les couples de clés d'un sous-ensemble aléatoire du réseau de confiance avec le même nombre de clés. Les distances sont calculées en utilisant l'ensemble du graphe, et non en se restreignant aux arêtes appartenant aux sous-graphes considérés. Les résultats expérimentaux semblent en adéquation avec la conjecture (voir tableau 1 p. 7) : la distance moyenne entre clés d'un même TLD est en général plus basse que celle entre clés aléatoires lorsque le TLD correspond à un pays.

On peut aussi penser que la distance entre les clés de deux TLD correspondant à des pays géographiquement proches devrait être plus basse que celle entre des TLD correspondant à des pays géographiquement éloignés.

Cependant, cette conjecture n'est pas validée par les résultats expérimentaux (voir tableau 2 p. 8). Une explication de ce phénomène est la difficulté que représente la comparaison des distances entre deux couples de TLD, puisque la structure individuelle de chaque TLD influe sur les résultats obtenus.

4 Représentation du graphe

4.1 Critères esthétiques

La représentation d'un graphe peut se faire suivant différents critères. Parmi les plus courants, citons (voir [1], p. 12-16) :

- Minimisation du nombre de croisements entre les arêtes (une solution sans croisement n'est possible que pour les graphes planaires).
- Respect d'une contrainte sur le rapport hauteur/largeur du dessin.
- Représentation des arêtes par des segments ayant autant que possible la même longueur.
- Dessin des arêtes avec des lignes aussi droites que possible.
- Maximisation de l'angle entre les représentations de deux arêtes incidentes à un même sommet.
- Respect des symétries.

Différentes approches générales peuvent être retenues pour le dessin. Par exemple, on peut décider de représenter les arêtes par des segments quelconques, ou par des successions de segments verticaux ou horizontaux².

Le critère esthétique retenu pour le dessin du réseau de confiance est la minimalité des variations entre la longueur des arêtes. En effet, l'objectif principal est l'étude des distances entre sommets, d'où la volonté de lier les distances sur la représentation aux distances dans le graphe. Les croisements n'ont que peu d'importance car les arêtes sont trop nombreuses pour être toutes représentées d'une manière lisible.

4.2 Choix d'un algorithme

Les algorithmes de dessin de graphe sont nombreux. Les différences entre eux concernent principalement les critères esthétiques qu'ils permettent de respecter, les types de graphe auxquels ils s'appliquent, et leurs performances. Un résumé est proposé par [1], p. 38.

L'algorithme force-directed a été retenu pour plusieurs raisons. Tout d'abord, il peut être appliqué à des graphes quelconques, au contraire d'autres algorithmes nécessitant des propriétés particulières que le graphe du réseau de confiance ne présente pas (caractère planaire, acyclique, etc.). Il suit également le critère esthétique choisi. Enfin, sa simplicité le rend assez performant en pratique.

4.3 Algorithme force-directed

L'algorithme force-directed modélise le graphe étudié comme un système physique, en considérant les arêtes comme des ressorts de longueur à vide fixée attachés à des masses représentant les sommets. À chaque itération, le système calcule la résultante des forces exercées sur chaque sommet (voir figure 8 p. 17) et le déplace légèrement dans la direction de la résultante. L'énergie potentielle des ressorts diminue au cours du temps, jusqu'à atteindre un minimum local qui est une position d'équilibre du système physique (et un dessin esthétiquement plaisant du graphe).

On ajoute habituellement une force de répulsion électrostatique entre les sommets pour éviter qu'ils ne s'entassent au centre. Cependant, le graphe du réseau de confiance est peu dense (le nombre total d'arêtes est petit devant le carré du nombre de sommets), donc le temps nécessaire au calcul de la force de répulsion, qui s'exerce entre tout couple de sommets, serait très grand devant celui nécessaire au calcul des forces de rappel des ressorts qui s'exercent pour chaque arête. Aussi, pour que les performances restent acceptables, le programme se limite au calcul des forces de rappel, ce qui permet d'avoir plusieurs itérations par seconde au lieu d'une itération au bout de quelques minutes.

Quelques adaptations ont dû être faites pour obtenir malgré tout une représentation exploitable. Afin de limiter la tendance à l'agglutinement au centre, les clés sont initialement disposées sur un cercle grand devant la longueur à vide des ressorts. Les clés se déplacent suivant la résultante des forces non pas d'un petit déplacement fixe, mais d'un déplacement aussi grand que nécessaire tant que cela contribue à la réduction de l'énergie potentielle ; cela semble empiriquement favoriser l'apparition d'alignements de clés en périphérie du graphe. Au

²De tels dessins sont utiles pour l'intégration à très grande échelle (VLSI), selon [2], p. 199.

contraire, certaines optimisations qui faisaient diminuer l'énergie potentielle plus vite ont dû être abandonnées car elles rendaient le graphe illisible...

4.4 Interface

Les bibliothèques `SDL`, `SDL.ttf`, `SDL.Input` et `SDL.Input.TTF` sont utilisées pour représenter le graphe au fur et à mesure de l'exécution de l'algorithme.

L'interface développée, outre l'affichage du graphe, offre de nombreuses commandes récapitulées dans le tableau 5 (p. 12).

4.5 Observations

Le phénomène de proximité entre clés d'un même TLD national dans le graphe s'observe aussi sur la représentation graphique après exécution de l'algorithme, comme le montrent les tableaux 3 p. 10 (avant l'exécution de l'algorithme) et 4 p. 11 (après l'exécution).

Lors de l'évolution du dessin, on observe que certaines clés mal intégrées restent en périphérie de la représentation graphique. Il s'agit en général de clés reliées au reste du réseau par un seul maillon. Ces clés sont le plus souvent membres du même TLD, voire du même domaine. Dans certains cas, toutes les clés d'un même domaine se retrouvent au même endroit sur le dessin (voir figures 9 p. 18, 10 p. 19, 11 p. 20, 12 p. 21 et 13 p. 22).

De manière générale, il y a une différence graphique observable à l'œil nu entre les ensembles de clés correspondant à un TLD national et les ensembles de clés sélectionnés aléatoirement ; les ensembles correspondant à un TLD national comprennent le plus souvent la totalité ou la quasi-totalité de plusieurs ensembles de clés en périphérie. On peut par exemple comparer la figure 5 (p. 14), qui met en évidence les clés autrichiennes, à la figure 6 (p. 15), qui met en évidence le même nombre de clés aléatoires.

Même vers le centre de la représentation, où des clés se retrouvent graphiquement proches bien qu'éloignées dans le graphe, on observe que la répartition des différents TLD n'est pas vraiment homogène, comme l'illustre la figure 7 p. 16.

Le logiciel permet aussi de repérer quelques curiosités du réseau de confiance. Voir par exemple l'image 13 p. 22.

5 Prolongements envisageables

Des améliorations de différents types pourraient être apportées au programme. Certains choix d'implémentation se sont révélés peu judicieux ; un bon nombre de fonctions pourrait être regroupé en fonctions génériques ; il faudrait à plusieurs endroits supprimer les limites stockées dans des constantes globales et utiliser `malloc`.

Pour l'ajout d'une force de répulsion électrostatique, l'utilisation de structures de données telles que des quadrees pourrait permettre de regrouper les clés selon leur position sur la représentation graphique. Ainsi, les effets de la répulsion pourraient être approximés en représentant les ensembles de clés éloignées de la clé d'étude par des masses ponctuelles pour réduire les calculs.

Une telle adaptation de l'algorithme *force-directed* est mise en œuvre par le projet FADE [12], qui affirme atteindre des temps d'exécution en $\Theta(n \log n)$.

Références

- [1] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, Ioannis G. Tollis, *Graph Drawing : Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, 1999.
- [2] Gary Chartrand, *Introductory Graph Theory*. Dover, New York, 1985.
- [3] Jean-Guillaume Dumas, Jean-Louis Roch, Éric Tannier, Sébastien Varrette, *Théorie des codes : Compression, cryptage, correction*. Dunod, Paris, 2007.
- [4] Alfred Aho, Jeffrey Ullman, *Concepts fondamentaux de l'informatique*. Trad. X. Cazin, I. Gourhant, J.-P. Le Narzul. Dunod, Paris, 1993.
- [5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, deuxième édition. MIT Press et McGraw-Hill, Cambridge, Massachusetts, 2001.
- [6] Wotsap [En ligne]. Jörgen Cederlöf, 2006. Disponible à l'adresse : <http://www.lysator.liu.se/~jc/wotsap/>
- [7] PGP pathfinder and key statistics [En ligne]. Henk P. Penning, 2009. Disponible à l'adresse : <http://pgp.cs.uu.nl/>
- [8] The Footsie Web of Trust analysis [En ligne]. Matthew Wilcox, 2009. Disponible à l'adresse : <http://www.parisc-linux.org/~willy/wot/footsie/>
- [9] Keyanalyse [En ligne]. M. Drew Streib, 2002. Disponible à l'adresse : <http://dtype.org/keyanalyze/>
- [10] PGP Web of Trust Statistics [En ligne]. Neal McBurnett, 1997. Disponible à l'adresse : <http://bcn.boulder.co.us/~neal/pgpstat/>
- [11] Sig2dot GPG/PGP Keyring Graph Generator [En ligne]. Nathaniel E. Barwell, 2002. Disponible à l'adresse : <http://www.chaosreigns.com/code/sig2dot/>
- [12] FADE [En ligne]. Aaron J. Quigley, 2006. Disponible à l'adresse : <http://www.csi.ucd.ie/staff/aquigley/home/?Research:Projects:FADE>
- [13] RFC 4880 [En ligne]. J. Callas, L. Donnerhacke, H. Finney, D. Shaw, R. Thayer, 2007. Disponible à l'adresse : <http://tools.ietf.org/html/rfc4880>
- [14] The Web of Trust .wot file format, version 0.2 [En ligne]. Jörgen Cederlöf, 2004. Disponible à l'adresse : <http://www.lysator.liu.se/~jc/wotsap/wotfileformat.txt>

TLD	Nombre clés	Distance TLD	Distance aléa.	Différence
TOTAL	41229	5.967422	5.967422	0
.de	10195	5.247043	5.966777	0.719734
.com	8793	6.435397	5.964490	-0.470907
.org	4633	5.228405	5.960977	0.732572
.net	4430	5.955349	5.895636	-0.059713
.edu	1855	6.211485	5.915838	-0.295647
.at	1126	6.549152	6.009203	-0.539949
.ch	938	4.812058	5.979436	1.167379
.uk	902	5.874658	5.885630	0.010972
.nl	807	4.832648	5.803330	0.970682
.se	616	5.126112	5.951894	0.825782
.fr	603	5.251495	5.920090	0.668595
.it	494	5.032766	6.019612	0.986846
.au	431	4.849732	5.807026	0.957295
.ca	403	5.612700	5.792844	0.180144
.no	266	5.907768	5.801345	-0.106422
.fi	245	5.428255	6.007280	0.579025
.es	213	4.822213	5.884172	1.061959
.dk	207	4.844570	6.057108	1.212537
.pl	196	5.430810	6.089103	0.658293
.cz	186	5.407966	5.947624	0.539658
.br	181	6.489210	6.159641	-0.329569
.be	181	5.078203	5.999817	0.921614
.nz	172	4.831362	6.060675	1.229313
.info	164	5.354030	5.617787	0.263757
.gov	151	6.435770	5.676988	-0.758783
.jp	149	5.757038	5.895996	0.138958
.name	110	4.895455	6.241157	1.345702
.hu	102	5.557478	5.784218	0.226740
.eu	93	4.972598	6.051914	1.079316
.us	78	5.574129	5.989809	0.415680
.ru	77	6.546298	5.444594	-1.101703
.gr	61	4.504434	5.700349	1.195915
.cx	60	4.619722	6.052222	1.432500
.mil	59	6.289572	5.888251	-0.401321
.ar	59	5.151393	6.166619	1.015226
.nu	55	4.975207	5.650248	0.675041
.ie	51	4.712034	6.086121	1.374087
.cl	49	4.020825	5.748855	1.728030
.cc	49	5.463140	5.857976	0.394835
.li	48	4.193576	5.577257	1.383681
.il	48	4.432292	6.670139	2.237847

TAB. 1 – Distance moyenne entre tout couple de clés pour chaque TLD, comparé aux distances pour un sous-ensemble aléatoire de clés de même taille. Les colonnes indiquent respectivement le nombre de clés dans le TLD, la distance moyenne entre tout couple de clés du TLD, la distance moyenne entre tout couple de clés du sous-ensemble aléatoire, et la différence de ces deux colonnes. Pour les TLD correspondant à un pays, la distance moyenne du TLD est en général plus basse que celle de l'ensemble de clés aléatoires.

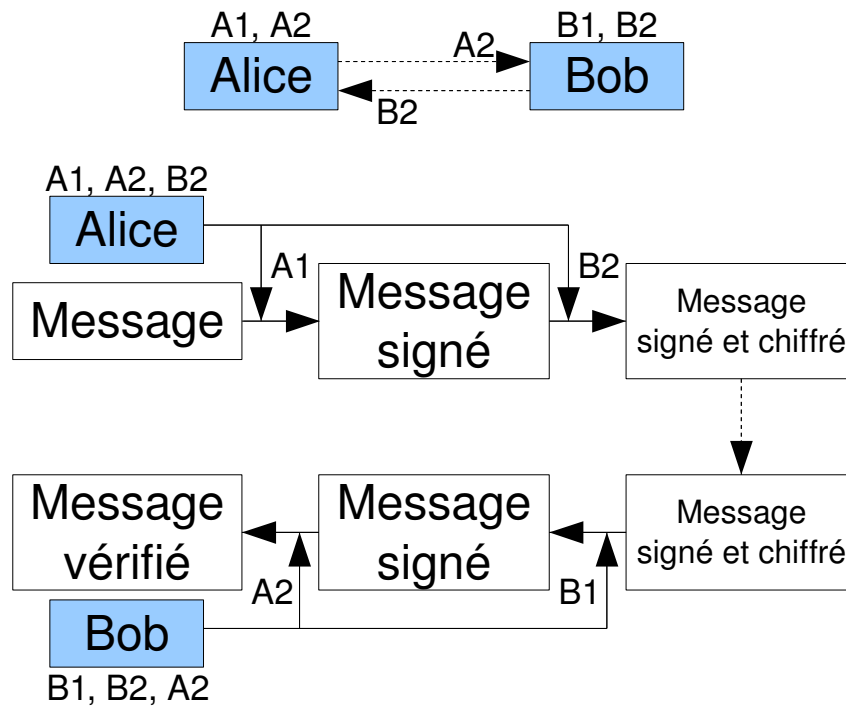


FIG. 1 – Schéma de principe de la communication à l'aide de la cryptographie asymétrique. Les clés A1 et A2 sont respectivement les clés privée et publique d'Alice, B1 et B2 celles de Bob. Les flèches en pointillés indiquent le transfert de données sur un canal vulnérable aux attaques passives. Le protocole assure à Alice que son message n'est lisible que par Bob, et assure à Bob l'authenticité du message reçu.

	.com	.de	.org	.edu	.uk	.net	.fr	.nl	.ch	.at	.au	.se	.ca	.it	
.com	6.43	6.25	6.00	6.53	6.41	6.22	6.28	6.02	6.06	7.22	6.03	6.50	6.13	6.26	.com
.de	5.92	5.27	5.43	6.20	5.92	5.62	5.71	5.44	5.28	6.41	5.61	6.09	5.77	5.69	.de
.org	5.72	5.48	5.21	5.87	5.66	5.49	5.43	5.25	5.28	6.44	5.26	5.80	5.44	5.44	.org
.edu	6.27	6.31	5.90	6.11	6.30	6.12	6.28	5.93	6.04	7.30	5.89	6.26	5.92	6.25	.edu
.uk	6.19	6.04	5.71	6.32	5.88	5.97	5.97	5.72	5.80	6.93	5.71	6.22	5.89	5.95	.uk
.net	6.18	5.89	5.71	6.32	6.14	5.94	5.97	5.72	5.72	6.87	5.79	6.26	5.92	5.96	.net
.fr	6.10	5.85	5.53	6.33	6.00	5.85	5.31	5.50	5.64	6.80	5.69	6.22	5.87	5.72	.fr
.nl	5.78	5.55	5.31	5.91	5.72	5.54	5.52	4.81	5.29	6.59	5.45	5.78	5.58	5.59	.nl
.ch	5.78	5.35	5.29	5.98	5.76	5.51	5.55	5.32	4.72	6.44	5.39	5.89	5.62	5.49	.ch
.at	7.59	7.11	7.05	7.86	7.47	7.30	7.28	7.10	6.92	6.72	7.23	7.68	7.38	7.28	.at
.au	5.84	5.75	5.36	5.95	5.76	5.64	5.66	5.48	5.49	6.63	4.87	5.85	5.44	5.63	.au
.se	6.13	6.08	5.73	6.16	6.12	5.95	6.08	5.73	5.83	7.03	5.74	5.15	5.84	6.00	.se
.ca	6.13	6.02	5.70	6.20	6.12	5.95	6.00	5.81	5.84	6.98	5.67	6.26	5.59	5.98	.ca
.it	6.11	5.85	5.56	6.34	6.04	5.86	5.75	5.62	5.58	6.81	5.64	6.17	5.88	4.98	.it
	.com	.de	.org	.edu	.uk	.net	.fr	.nl	.ch	.at	.au	.se	.ca	.it	

TAB. 2 – Distance moyenne entre les clés d'un TLD et celles d'un autre TLD. Aucune tendance notable ne semble pouvoir être observée. Noter que le tableau n'est pas symétrique, car le graphe du réseau de confiance est orienté. Les distances sont indiquées en partant du TLD de la ligne pour aller jusqu'au TLD de la colonne.

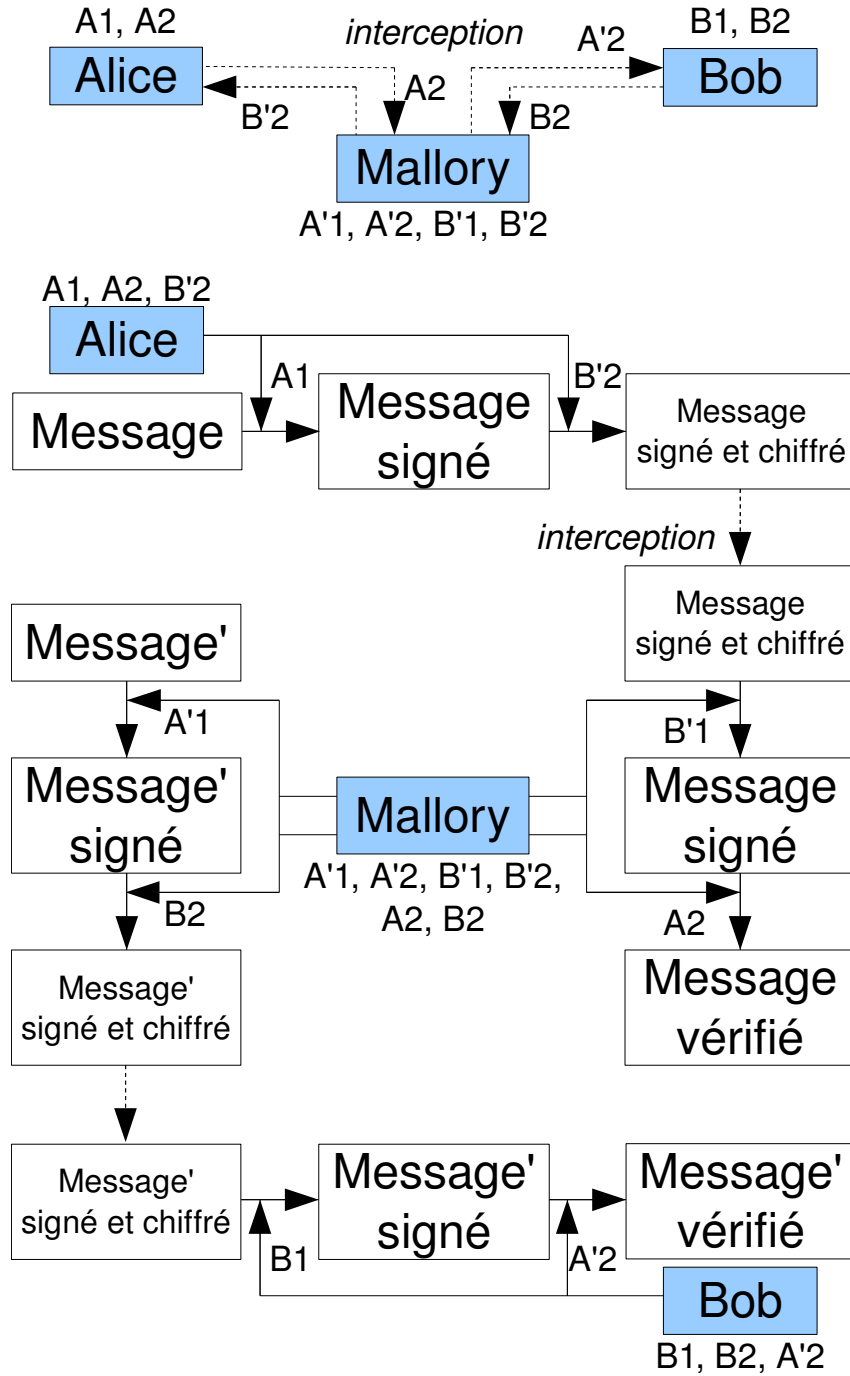


FIG. 2 – Schéma de principe de l'attaque de l'homme du milieu, menée par Mallory. Les clés A1 et A2 sont respectivement les clés privée et publique d'Alice, B1 et B2 celles de Bob, et A'1, A'2, B'1, B'2 des clés factices créées par Mallory. Les flèches en pointillés indiquent le transfert de données sur un canal vulnérable aux attaques actives. En se faisant passer pour Bob auprès d'Alice et pour Alice auprès de Bob, Mallory est en mesure de lire et de modifier le message.

TLD	Nombre	Dist. graph. TLD	Dist. graph. rand	Diff. graph.
TOTAL	41518	2734218.640580	2734218.640580	0.000000
.de	10293	2734129.696247	2734177.930927	48.234679
.com	8869	2733828.170516	2733933.050529	104.880013
.org	4633	2733977.717087	2733715.849931	-261.867157
.net	4456	2733992.641092	2732461.801724	-1530.839368
.edu	1825	2731891.350734	2732554.094312	662.743579
.at	1163	2731582.253503	2732115.958488	533.704984
.ch	984	2731321.516533	2732120.941978	799.425445
.uk	903	2732030.791044	2732709.621568	678.830524
.nl	831	2728158.928604	2732165.022434	4006.093830
.se	617	2730642.992385	2731590.759039	947.766655
.fr	609	2729937.588843	2724927.897378	-5009.691465
.it	480	2730961.945102	2724485.792345	-6476.152756
.au	427	2731317.671051	2722117.294296	-9200.376755
.ca	399	2731345.247024	2725666.015186	-5679.231837
.no	266	2728807.302787	2717153.803587	-11653.499200
.fi	248	2721836.257568	2722882.247815	1045.990247
.es	226	2726391.307467	2725909.281392	-482.026075
.dk	202	2725406.693791	2723440.845521	-1965.848270
.pl	199	2701674.611850	2719000.794424	17326.182575
.cz	192	2718595.437915	2695349.156512	-23246.281402
.be	186	2730342.275110	2727246.037406	-3096.237704
.br	182	2695093.651967	2715819.790229	20726.138262
.nz	172	2714471.658352	2726946.900561	12475.242209
.info	158	2722016.231088	2711691.334743	-10324.896344
.gov	155	2709476.691159	2717992.148792	8515.457633
.jp	145	2726965.567007	2715782.797165	-11182.769843
.name	111	2709557.502597	2721646.929417	12089.426820
.hu	103	2674100.096610	2697567.239409	23467.142799
.eu	97	2697815.814930	2694774.577530	-3041.237400
.us	79	2682433.685950	2713789.859772	31356.173822
.ru	75	2715675.548525	2717471.600472	1796.051947
.cx	61	2693122.809399	2683110.290740	-10012.518659
.gr	59	2695408.599987	2705460.704631	10052.104643
.ar	59	2690992.924995	2683882.886890	-7110.038105
.mil	58	2624351.626181	2672674.391637	48322.765456
.nu	57	2672927.687521	2671022.354921	-1905.332600
.ie	54	2719083.826405	2697081.308993	-22002.517412
.cc	54	2711953.537288	2645641.015326	-66312.521962
.li	52	2663439.484475	2694137.930846	30698.446371
.cl	51	2685184.329081	2721458.757083	36274.428002
.il	48	2712402.947470	2702129.847129	-10273.100341

TAB. 3 – Distance moyenne graphique (euclidienne) entre tout couple de clés pour chaque TLD, comparé aux distances pour un sous-ensemble aléatoire de clés de même taille, avant lancement de l’algorithme force-directed. Les colonnes sont les mêmes que celles du tableau 1, à ceci près qu’il s’agit ici de distances graphiques. Aucune tendance notable ne semble pouvoir être observée.

TLD	Nombre	Dist. graph. TLD	Dist. graph. rand	Diff. graph.
TOTAL	41518	3193.064371	3193.064371	0.000000
.de	10293	2506.366864	3230.252117	723.885253
.com	8869	3164.285610	3140.449526	-23.836084
.org	4633	5042.325938	3263.105052	-1779.220886
.net	4456	2815.622084	3318.198797	502.576714
.edu	1825	4300.806350	3538.412800	-762.393550
.at	1163	2271.667924	2959.093469	687.425545
.ch	984	2428.502339	3219.626131	791.123791
.uk	903	2717.983256	3029.249258	311.266002
.nl	831	1944.275874	2944.523480	1000.247606
.se	617	2172.339745	3034.948489	862.608744
.fr	609	2163.335576	2747.032359	583.696783
.it	480	1907.802628	2790.056523	882.253896
.au	427	1906.782654	2845.420643	938.637989
.ca	399	2166.010990	2929.184160	763.173169
.no	266	4132.550155	3990.157557	-142.392598
.fi	248	2241.706746	2664.099467	422.392721
.es	226	2387.264135	3277.433316	890.169181
.dk	202	2278.013934	3010.094538	732.080604
.pl	199	2010.590806	3112.368173	1101.777366
.cz	192	2919.197356	3721.482760	802.285404
.be	186	1782.686669	2534.432093	751.745423
.br	182	2758.758147	3556.436895	797.678748
.nz	172	1540.284712	3639.189005	2098.904293
.info	158	2202.237291	3126.149543	923.912252
.gov	155	2853.553849	3419.255919	565.702070
.jp	145	2729.735426	3254.006170	524.270744
.name	111	1884.525307	2942.834560	1058.309253
.hu	103	1585.385901	3370.923366	1785.537465
.eu	97	2325.132918	4086.783790	1761.650871
.us	79	2048.731833	2891.677212	842.945379
.ru	75	2068.142556	3160.869485	1092.726929
.cx	61	1612.862343	2588.744234	975.881891
.gr	59	2234.572861	2811.852891	577.280030
.ar	59	3661.775134	4416.189455	754.414321
.mil	58	2572.962541	2978.316256	405.353714
.nu	57	2088.583784	2259.275753	170.691968
.ie	54	1681.149227	2430.957943	749.808716
.cc	54	2424.158831	2967.629716	543.470885
.li	52	2266.359168	2441.486862	175.127694
.cl	51	1457.802389	2556.356192	1098.553804
.il	48	1749.870133	1890.371893	140.501760

TAB. 4 – Distance moyenne graphique (euclidienne) entre tout couple de clés pour chaque TLD, comparé aux distances pour un sous-ensemble aléatoire de clés de même taille, après exécution de l’algorithme force-directed pendant quelques heures. Les colonnes sont les mêmes que celles du tableau 3. Pour les TLD correspondant à un pays, la distance moyenne du TLD est en général plus basse que celle de l’ensemble de clés aléatoires.

Entrée	Effet
Clic gauche	Sélection de clé(s)
Clic droit	Déplacement de clé(s)
Clic central	Déplacement de la vue
Molette	Zoom
a	Tout sélectionner
c	Colorier les clés
d	Calculer les distances entre clés
e	Sélection par TLD ou adresse de courriel
f	Marquage des clés de départ pour les calculs de distance
g	Affichage de la résultante des forces
i	Affichage des identifiants de clés
k	Sélection par identifiant
l	Recalcul manuel de la résultante
m	Déplacement manuel suivant la résultante
n	Affichage des noms et adresses de courriel
q	Quitter
r	Sélection de clés aléatoires
s	Sélection des clés ayant signé les clés sélectionnées
t	Affichage du nombre de clés dans la sélection
v	Inversion de la sélection
x	Activation ou désactivation de l'algorithme force-directed
z	Zoom automatique
/	Remise à zéro des opérateurs
+	Opérateur union
-	Opérateur différence ensembliste
*	Opérateur intersection
\	Opérateur différence symétrique
A	Tout sélectionner
C	Coloriage rapide
D	Suppression des marques de départ et d'arrivée
F	Marquage des clés d'arrivée pour les calculs de distance
G	Masquage de la résultante des forces
I	Masquage des identifiants de clés
L	Recalcul manuel de la résultante (toutes les clés)
M	Déplacement manuel suivant la résultante (toutes les clés)
N	Masquage des noms et adresses de courriel
S	Sélection des clés ayant été signées par les clés sélectionnées
Z	Centrer la vue
Ctrl+A	Calcul de données pour les tableaux 1, 3 et 4
Ctrl+B	Calcul de données pour le tableau 2
Ctrl+C	Arrangement des clés sélectionnées en cercle

TAB. 5 – Liste des commandes du logiciel.

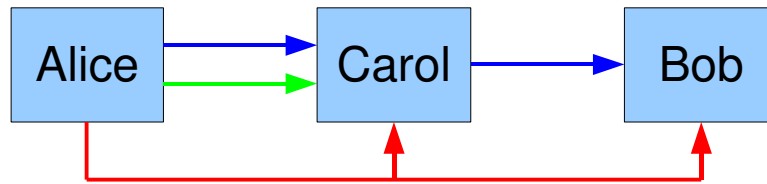


FIG. 3 – Schéma de principe du réseau de confiance. Les flèches bleues, vertes et rouges indiquent la signature de clé, la confiance en une personne, et l’assurance de la validité de la clé respectivement. Alice a vérifié la clé de Carol (elle a donc signé la clé de Carol) et a confiance en Carol, et Carol a vérifié la clé de Bob (elle a donc signé la clé de Bob), donc Alice a une garantie de la validité de la clé de Bob.

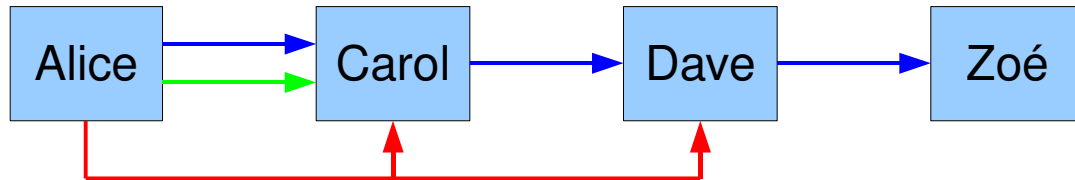


FIG. 4 – Schéma de principe de la non-transitivité du réseau de confiance. La légende est celle de la figure 3. Alice a vérifié la clé de Carol et a confiance en elle, et celle-ci a vérifié la clé de Dave. Alice a donc une garantie de la validité de la clé de Dave, mais pas de celle de Zoé puisqu’Alice n’a pas confiance en Dave. Voir [3], p. 209 et 312.

Listing 1 – main.c : Fonction main

```

1  #include "main.h"
2
3
4  // array of keys
5  key vertices[MAXKEYS];
6  unsigned long num_keys = 0;
7
8  // array of signatures
9  struct sig sigs[MAXSIGS];
10 unsigned int num_sigs=0;
11
12 // parameters for graphical interface
13 char auto_recalc=0;
14 char select_mode=SELECT.SET;
15 int per_pass=PER.PASS;
16
17
18 int main(int argc, char **argv)
19 {
20     int rsl;
21     char msg[500];
22
23     // check command line parameters
24     check(argc, argv);
25     // initialise globals
26     globals_init();
27     // load data file
28     load(argv[1]);
29     // initialise graphics
30     graphics_init();
31     // initialise display
32     reset_frame();
33
34     sprintf(msg, "%lu keys loaded, %lu signatures.", num_keys, num_sigs);
35     help(msg);
36
37     while (1)
38     {
39         // redraw all keys
40         redraw_all(update_eps(0));
41
42         // improve key placement with force-directed algorithm
43         if (auto_recalc)
44             move_one_pass(STEP, per_pass);
45
46         // manage user events

```

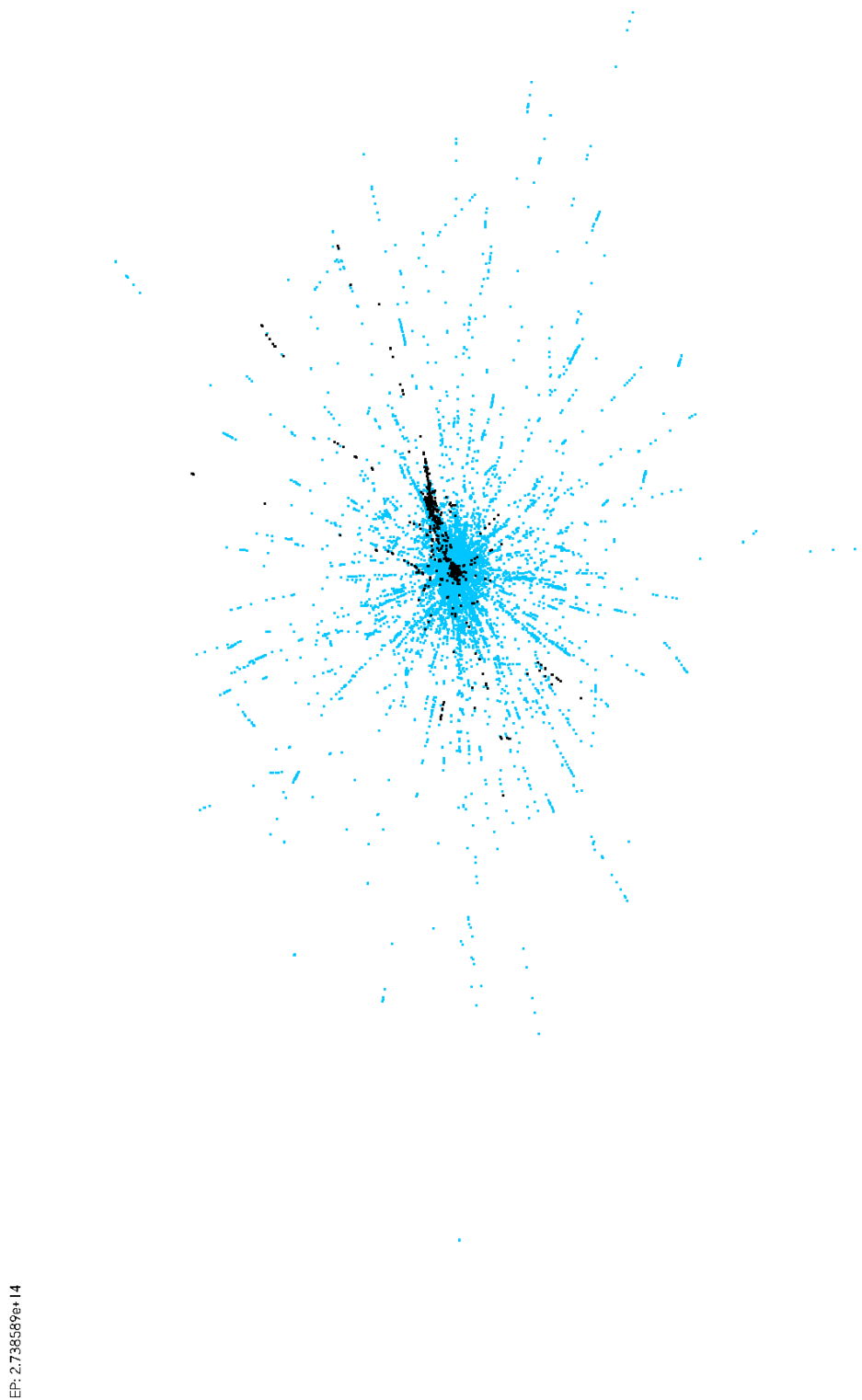


FIG. 5 – Position des clés autrichiennes (en noir) dans le réseau de confiance.

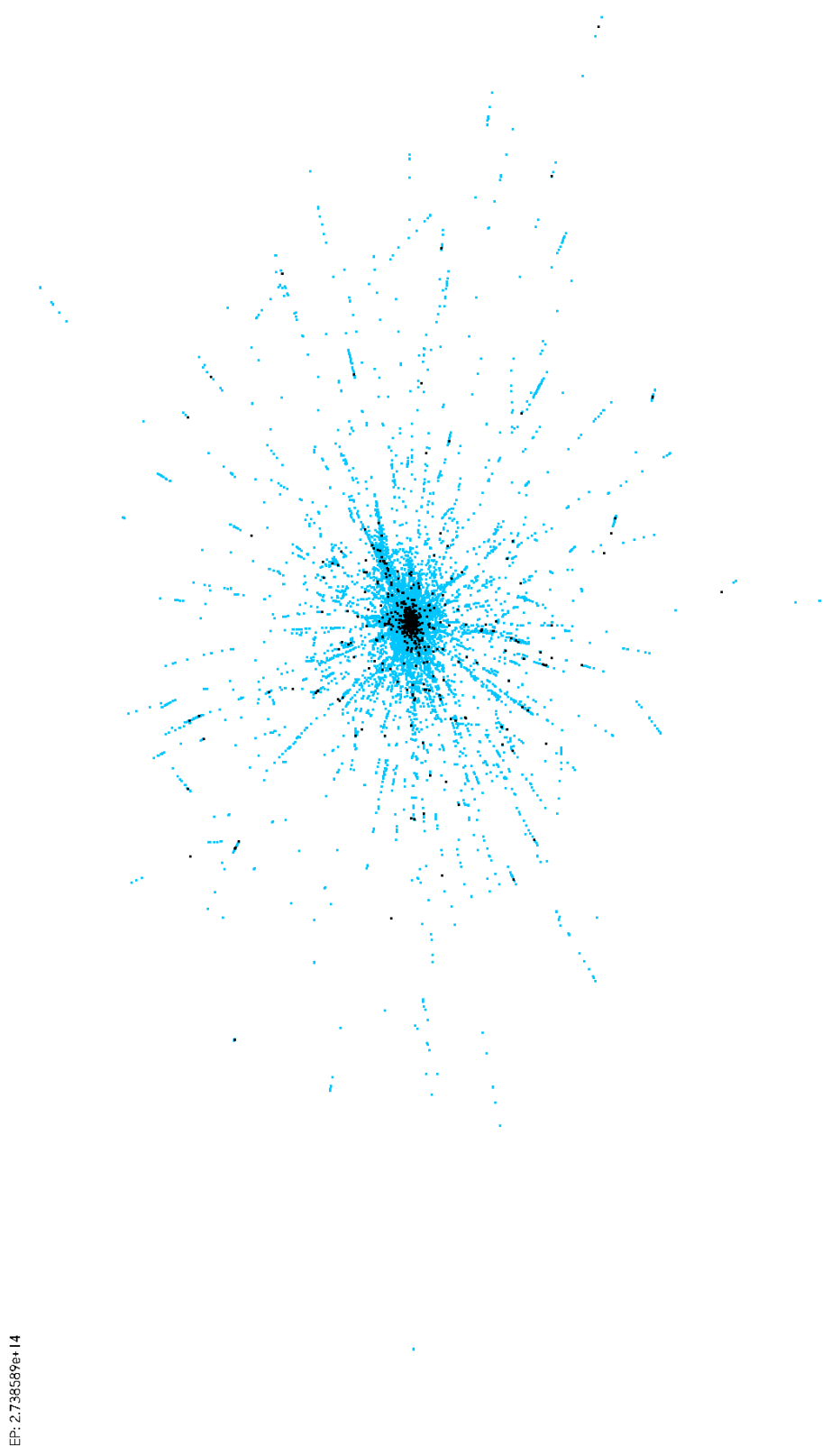


FIG. 6 – Position d'un sous-ensemble de clés aléatoires aussi nombreuses que les clés autrichiennes, à comparer avec la figure 5. On remarque que les clés autrichiennes ont davantage tendance à former des alignements.

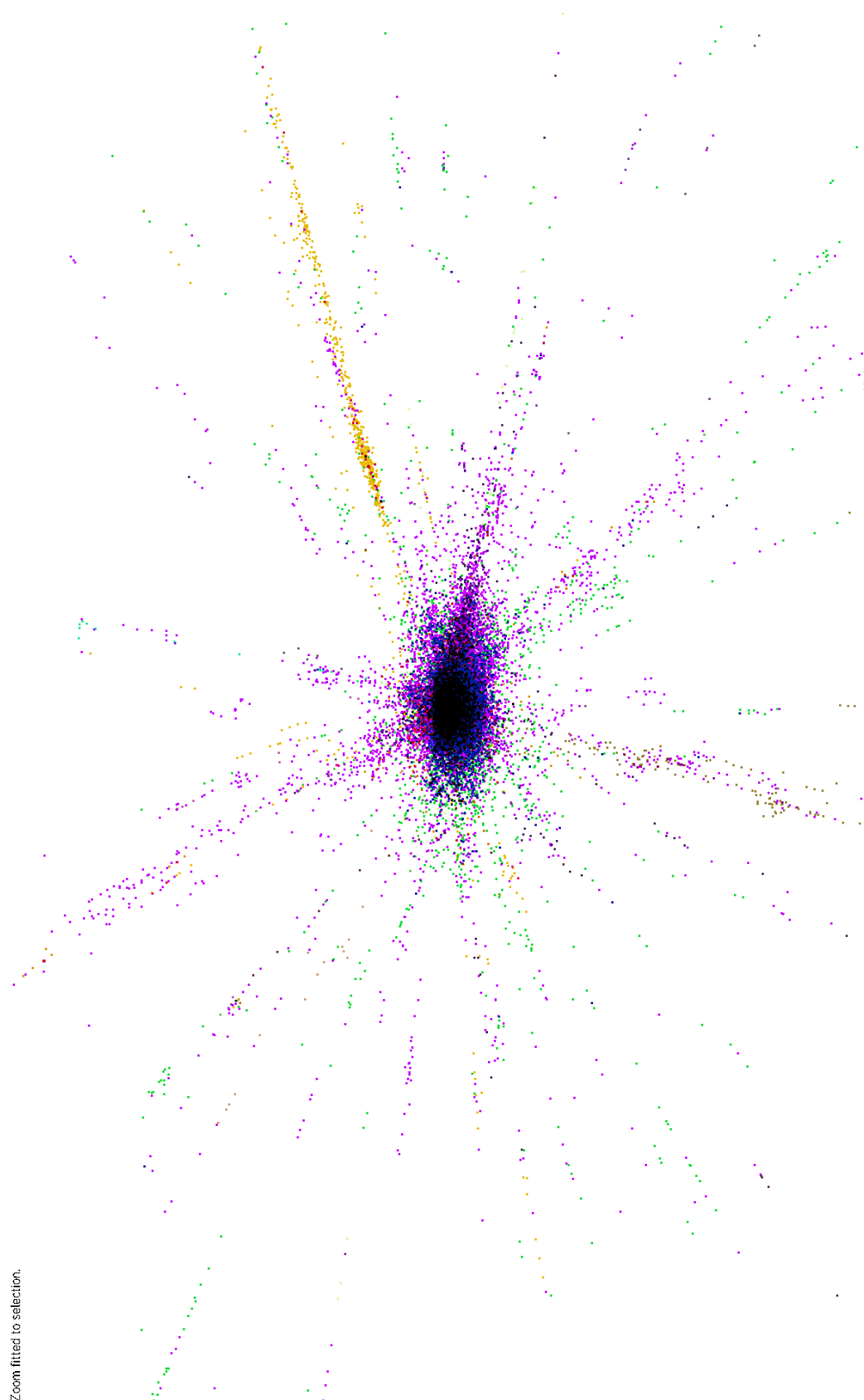


FIG. 7 – Centre du réseau de confiance, avec coloriage des clés selon leur TLD. On observe que les TLD ne sont pas répartis de façon homogène.

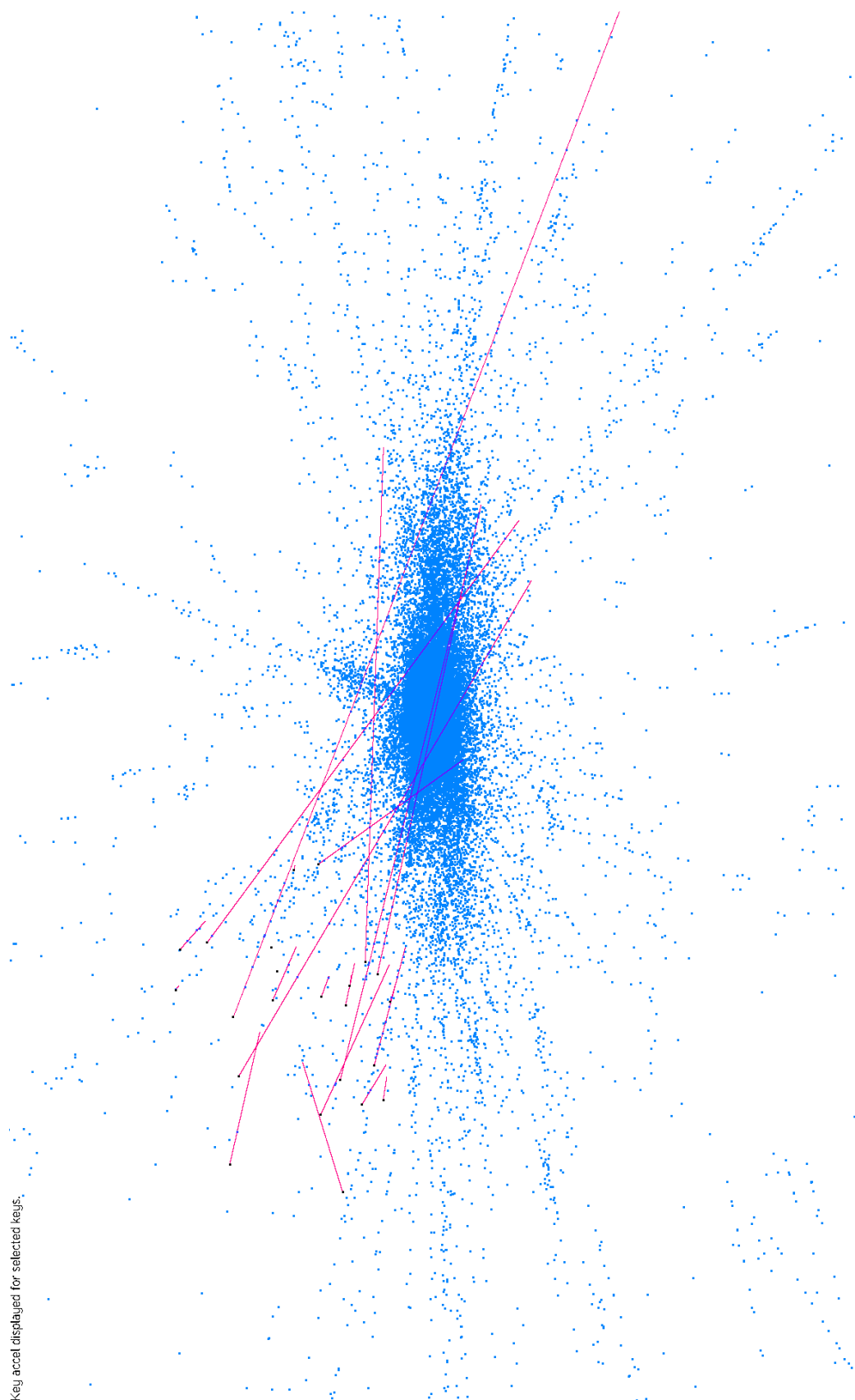


FIG. 8 – Affichage de la résultante des forces de rappel pour quelques sommets aléatoires.

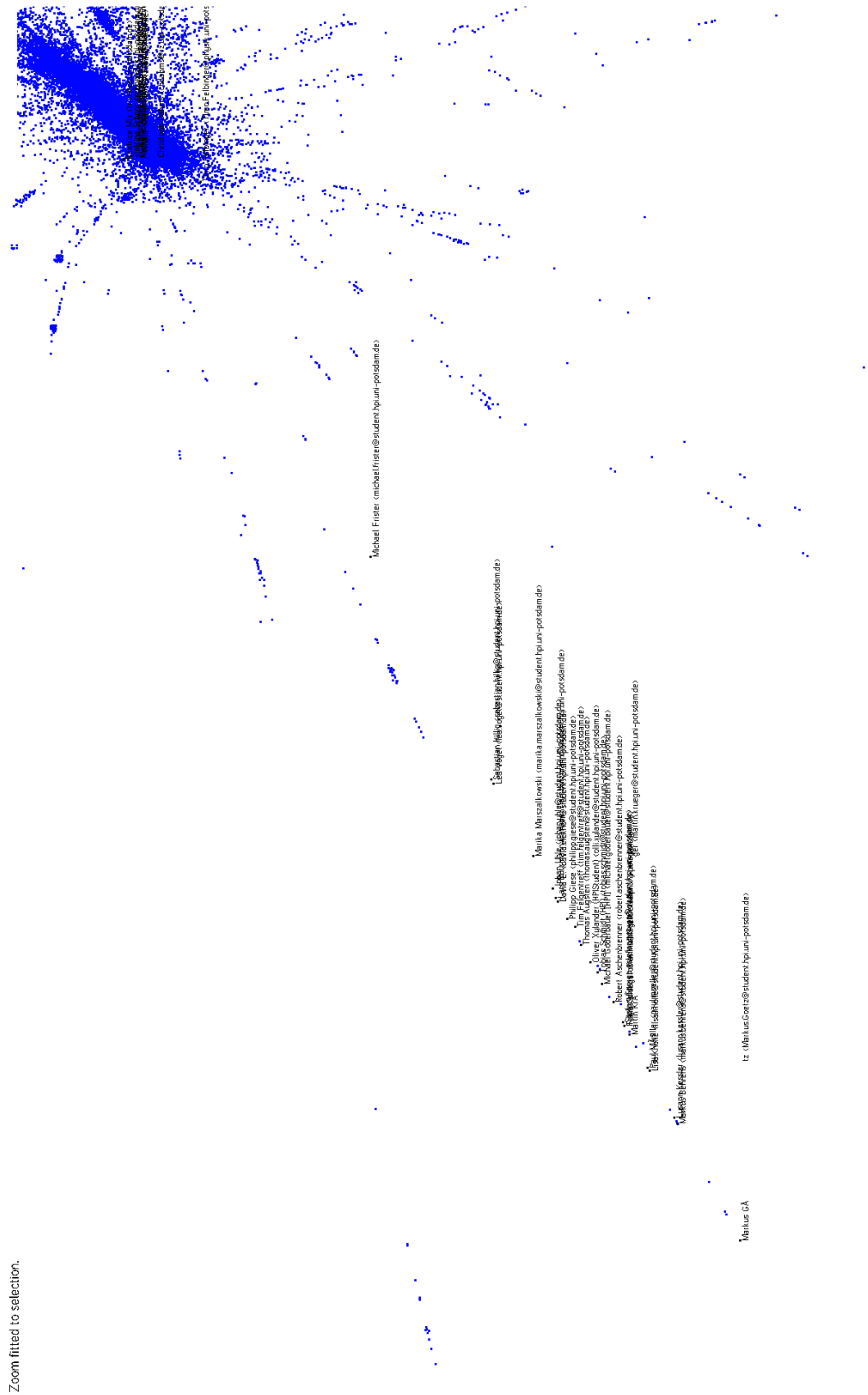


FIG. 9 – Ensemble des clés du nom de domaine `uni-potsdam.de` (Universität Potsdam), qui sont presque toutes au même endroit sur la représentation graphique.

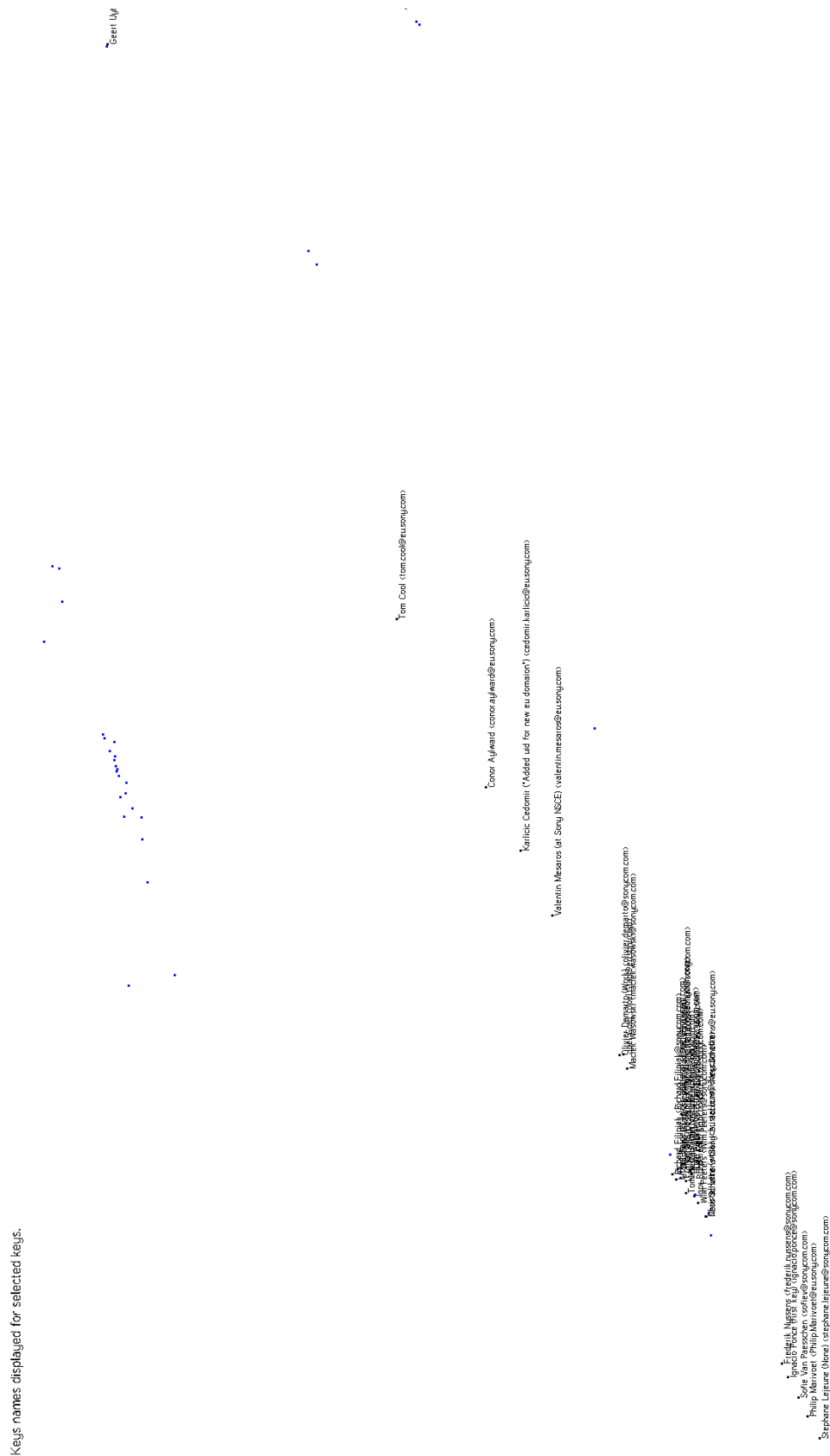


FIG. 10 – Ensemble des clés de noms de domaines associés à l'entreprise Sony, qui sont toutes au même endroit sur la représentation graphique.

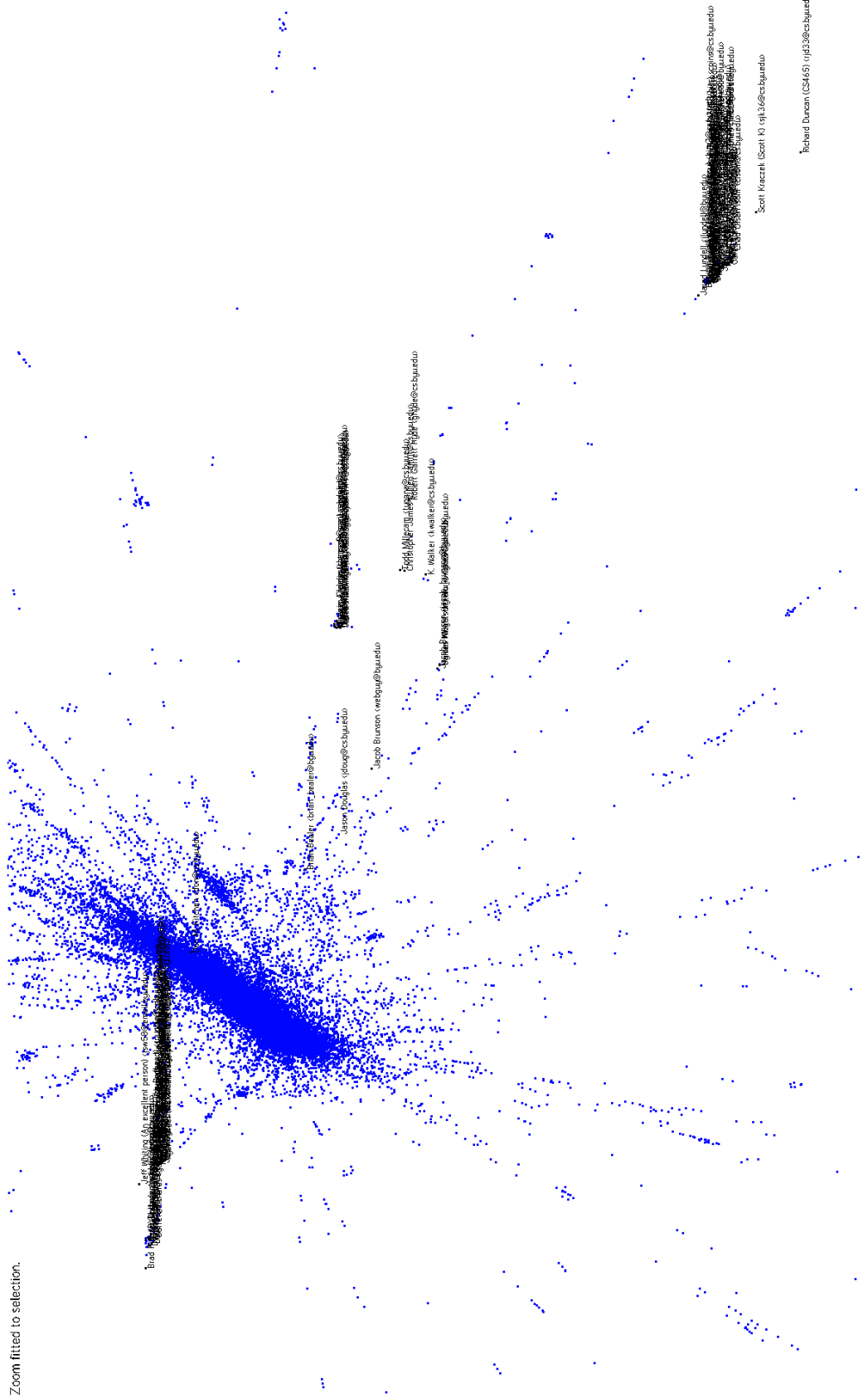


FIG. 11 – Ensemble des clés du nom de domaine `byu.edu` (Brigham Young University), qui sont presque toutes au même endroit sur la représentation graphique.

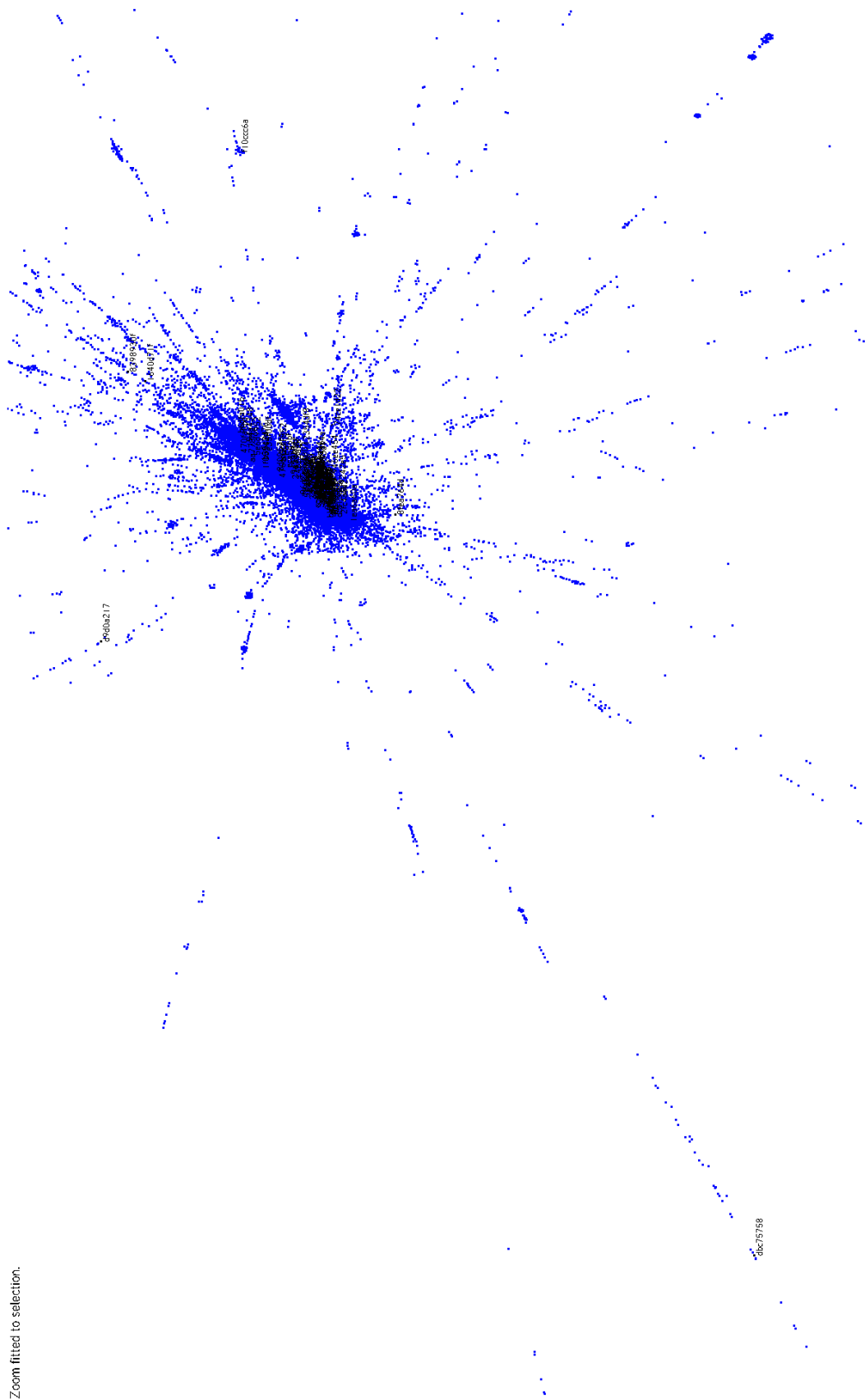


FIG. 12 – Ensemble de clés aléatoires aussi nombreuses que les clés du domaine `byu.edu` (à comparer avec la figure 11).



```
47     while ((rsl = manage_event()) > 0);
48     // exit if user requests it
49     if (rsl < 0) break;
50 }
51
52 // unload graphics
53 graphics_end();
54
55 return 0;
56 }
```

Listing 2 – main.h : Variables et constantes globales

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <math.h>
5  #include <SDL/SDL.h>
6  #include <SDL/SDL_ttf.h>
7  #include "SDL_Input/SDL_Input.h"
8  #include "SDL_Input/SDL_Input_TTF/SDL_Input_TTF.h"
9
10 // Limits. TODO: use malloc()!
11 #define MAXKEYS 50000
12 #define MAXNAMELEN 500
13 #define MAXSIGSIGS 50000
14 #define MAXTSIGS 500000 // total number of sigs
15
16 // Fine-tuning.
17 // TODO: allow the user to change these at runtime.
18 #define PER_PASS 10000 // how many keys to update per pass
19 #define MIN_MOVED PER_PASS/20
20 #define K1 1 // spring constant for energy
21 #define K2 1 // spring constant for pull
22 #define L0 1000 // spring default length
23 #define F 0.005 // accel factor
24 #define STEP 0.001 // move step
25 #define MULT_STEP 1.01 // increase in move step (to avoid getting stuck on a key)
26 #define DISPRATE 100 // how often shall we output status info
27 #define TLD_THRESHOLD 10 // min keys in TLD to take it into account
28
29 // Marks are binary masks which can be applied to vertices. They are used as a naive way to keep
30 // track of key subsets or key options.
31 #define MNONE 0 // no mark
32 #define MF 1 // mark from: compute distances from these keys
33 #define MT 2 // mark to: compute distances to these keys
34 #define MFT 3 // mark from to
35 #define MS 4 // mark seen: the key or TLD has already been done
36 #define MU 8 // user mark (selection): the key is in user selection
37 #define MV 16 // instant user mark (result of last selection command)
38 #define MI 32 // display key id
39 #define MN 64 // display key name
40 #define MA 128 // display accel
41 #define MALL 255
42
43 enum operator { SELECT_SET, SELECT_UNION, SELECT_INTERSECTION, SELECT_REMOVE, SELECT_XOR };
44
45 enum error { ERR_NONE, ERR_SYNTAX, ERR_TEMPDIR, ERR_BUNZIP, ERR_AR, ERR_QOF, ERR_QUF,
46             ERR_SDL_INIT, ERR_TTF_INIT, ERR_FONT, ERR_SDL_INPUT_INIT, ERR_VIDEO_INIT };
47
48 enum graph_color { BLACK, GRAY, WHITE }; // for BFS
49
50 enum coordinate { X, Y };
51
52 // FORWARDS is from signer to signee
53 // BACKWARDS is from signee to signer
54 enum graph_orientation { FORWARDS, BACKWARDS };
55
56 enum tooltip { TOOLTIP_NONE, TOOLTIP_HELP, TOOLTIP_INPUT };
57
58 // these are constants, but we don't initialise them right away
59 SDL_Color black;
60 SDL_Color white;
61 SDL_Color red;
62 SDL_Color green;
63 SDL_Color blue;
64
65 #define WIDTH 1280
66 #define HEIGHT 800
67 #define DEPTH 32
68
69 #define FONT "fonts/Tuffy.ttf"
70 #define INTERFACE_FONT_SIZE 18
71 #define KEY_FONT_SIZE 12
72
73 #define ZFACTOR 10.0 // zoom speed with mouse wheel
74
75 // key size on screen
76 #define DOT_SIZE 2
77
78 // selection rectangle size
79 #define SELECTION_RECT_SIZE 2

```



```
80 // margin around the keys when fitting view
81 #define FITOFFSET 0.1
82 // added to dimensions when fitting view
83 #define KEYSIZE 10000
84
85 // minimum distance for key selection (in pixels, more or less)
86 #define MIN_DIST 200
87
88 // difference in pixels to distinguish click and multi-select
89 #define SELECTION_DELTA 20
90
91 #define TOOLTIP_TIME 100
92
93 // adjust size of graph representation
94 #define RAND_DIVIDE_FACTOR 1000.0
95
96
97 typedef struct {
98     // TODO weird, should be unsigned long
99     unsigned int id;
100     char name[MAXNAMELEN];
101     char mark;
102     double p[2]; // position on planar representation
103     double a[2]; // acceleration on planar representation
104     SDL_Color color;
105 } key;
106
107
108 struct signature {
109     // a member of a chained list of signatures
110     struct sig * s; //the real signature
111     struct signature * next;
112 };
113
114 struct sig {
115     // a signature
116     unsigned long f; //from: signer
117     unsigned long t; //to: signee
118     char type;
119     double ep; // potential energy for planar representation
120     unsigned long id;
121 };
122
123 typedef struct {
124     // a chained list of signatures
125     unsigned long num; //useless
126     struct signature * head;
127     struct signature * last; //should be useless
128 } signatures;
129
130 typedef struct {
131     unsigned long q[MAXSIGS];
132     unsigned long h; // head pos
133     unsigned long t; // tail pos
134 } queue;
135
136
137
138 int id(int a);
139
140 void update_key_accel(unsigned long pos);
141 void move_key_step(unsigned long i);
142
143 void opsel_by_color(char*);
144 void opsel_by_id(char*);
145 void opsel_by_tld(char*);
146 void opsel_rand(char*);
147
148 void moving_adjust(unsigned long pos);
149 void opsel_by_region_p(double x1, double y1, double x2, double y2);
150
151 double disp2real_x(double disp_x);
152 double disp2real_y(double disp_y);
153 double sqrt(double x);
```

Listing 3 – load.c : Fonctions d’initialisation

```

1  #include "main.h"
2
3
4  extern key vertices[MAXKEYS];
5
6  extern struct sig sigs[MAXSIGS];
7  extern unsigned int num_sigs;
8  extern unsigned long num_keys;
9
10 extern char colors[MAXKEYS];
11 extern char depths[MAXKEYS];
12
13 extern signatures edges[MAXKEYS];
14 extern signatures redges[MAXKEYS];
15
16 extern void circle(unsigned long pos, double radius, double cx, double cy);
17
18 int id(int a) { return a; }
19
20 char check(int argc, char** argv)
21 {
22     // Check options syntax.
23
24     char* base_name = 0;
25
26     if ((base_name = strchr(argv[0], '/'))
27         base_name++;
28
29     if (argc == 1)
30     {
31         printf("Usage: base_name WOTFILE\n");
32         return ERR_SYNTAX;
33     }
34
35     return 0;
36 }
37
38 int create_temp_dir() { return system("mkdir -p temp"); }
39
40 int run_ar() { return system("cd temp; ar x wot_data"); }
41
42 enum error prepare(char* input)
43 {
44     // Extract files from the Wotsap file.
45
46     try(&create_temp_dir, &id, ERR_TEMPDIR, "Error: Cannot create temp folder.\n");
47
48     char command[500];
49     sprintf((char*) &command, "bunzip2 -cd %s > temp/wot_data", input);
50
51     if (system(command))
52     {
53         fprintf(stderr, "Error: Cannot bunzip input file.\n");
54         return ERR_BUNZIP;
55     }
56
57     try(&run_ar, &id, ERR_AR, "Error: ar did not succeed.\n");
58
59     return ERR_NONE;
60 }
61
62
63 void read_sig(FILE *f, unsigned long signer, unsigned long sig_data)
64 {
65     // Load a signature.
66
67     struct signature *new_sig;
68
69     new_sig = malloc(sizeof(struct signature));
70     new_sig->next = NULL;
71     new_sig->s = &sigs[num_sigs];
72     push(&(edges[num_keys]), new_sig);
73 }
74
75 void redge_init(unsigned long i)
76 {
77     redges[i].head = NULL; redges[i].last = NULL;
78 }
79
80 void reverse()
81 {

```

```

82 // Create reverse adjacency lists (by reversing existing lists).
83
84 unsigned long i;
85 struct signature *s, *new_sig;
86
87 do_all(&redges_init);
88 for (i=0; i<num_keys; i++)
89 {
90     s = edges[i].head;
91
92     while (s)
93     {
94         redges[s->s->f].num++;
95         new_sig = malloc(sizeof(struct signature));
96         new_sig->s = s->s;
97         new_sig->next = NULL;
98         push(&(redges[s->s->f]), new_sig);
99         s = s->next;
100     }
101 }
102 }
103
104 enum error load(char* input)
105 {
106     unsigned long i;
107
108     FILE *f_debug, *f_keys, *f_names, *f_readme, *f_signatures, *f_wotversion;
109
110     unsigned int sig_data;
111
112     // TODO check for errors
113     char ret=0;
114     ret = prepare(input);
115     if (ret) return ret;
116
117     f_debug = fopen("temp/debug", "r");
118     f_keys = fopen("temp/keys", "r");
119     f_names = fopen("temp/names", "r");
120     f_readme = fopen("temp/README", "r");
121     f_signatures = fopen("temp/signatures", "r");
122     f_wotversion = fopen("temp/WOTVERSION", "r");
123
124     while (!feof(f_signatures) && !feof(f_names) && !feof(f_keys))
125     {
126         vertices[num_keys].id = 0;
127         vertices[num_keys].id=read_ul(f_keys);
128         fgets(vertices[num_keys].name, MAXNAMELEN, f_names);
129         circle(num_keys, RAND_MAX/RAND_DIVIDE_FACTOR, RAND_MAX, RAND_MAX);
130         vertices[num_keys].color = blue;
131         trim_trailing_newline(num_keys);
132         edges[num_keys].num=read_ul(f_signatures);
133         edges[num_keys].head = NULL;
134         edges[num_keys].last = NULL;
135         rstm(num_keys);
136
137         for (i=0; i<edges[num_keys].num; i++)
138         {
139             sig_data = read_ul(f_signatures);
140             sigs[num_sigs].t = num_keys;
141             sigs[num_sigs].f = get_id_from_sig_data(sig_data);
142             sigs[num_sigs].type = get_type_from_sig_data(sig_data);
143             sigs[num_sigs].ep = 0;
144             sigs[num_sigs].id = num_sigs;
145             read_sig(f_signatures, num_keys, sig_data);
146             num_sigs++;
147         }
148         num_keys++;
149     }
150
151     // slight offset problem
152     num_keys--;
153
154     reverse();
155
156     fclose(f_debug);
157     fclose(f_keys);
158     fclose(f_names);
159     fclose(f_readme);
160     fclose(f_signatures);
161     fclose(f_wotversion);
162
163     // TODO: cleanup temp dir
164     return ERR_NONE;
165 }

```

Listing 4 – structures.c : Structures fondamentales

```

1  #include "main.h"
2
3  extern key vertices[MAXKEYS];
4
5  extern struct sig sigs[MAXSIGS];
6  extern unsigned int num_sigs;
7  extern unsigned long num_keys;
8
9  char colors[MAXKEYS];
10 char depths[MAXKEYS];
11
12 extern signatures edges[MAXKEYS];
13 extern signatures redges[MAXKEYS];
14
15 void enqueue(unsigned long pos, queue* q)
16 {
17     q->q[q->t] = pos;
18     q->t++;
19     if (q->t == MAXSIGS)
20         q->t = 0;
21     if (q->h == q->t)
22     {
23         printf("Error: queue overflow.\n");
24         exit(ERR_QOF);
25     }
26 }
27
28 unsigned long dequeue(queue* q)
29 {
30     unsigned long rsl;
31     if (q->h == q->t)
32     {
33         printf("Error: queue underflow.\n");
34         exit(ERR_QUF);
35     }
36     rsl = q->q[q->h];
37     q->h++;
38     if (q->h == MAXSIGS)
39         q->h = 0;
40     return rsl;
41 }
42
43 int queue_empty(queue* q)
44 {
45     return q->h==q->t;
46 }
47
48 void queue_reset(queue* q)
49 {
50     q->h=q->t=0;
51 }
52
53
54 char is_empty(signatures s)
55 {
56     return s.head==NULL;
57 }
58
59
60 void push(signatures *stack, struct signature *s)
61 {
62     if (is_empty(*stack))
63     {
64         stack->last = s;
65         stack->head = s;
66     } else {
67         stack->last->next = s;
68         stack->last = s;
69     }
70 }
71
72
73 int next(struct sig* s, int orientation)
74 {
75     // Find next key, according to orientation.
76
77     if (orientation == FORWARDS)
78     {
79         return s->t;
80     } else {
81         return s->f;

```

```
82     }
83 }
84
85 void breadth_explore_init(unsigned long i)
86 {
87     colors[i] = WHITE;
88     depths[i] = -1;
89 }
90
91
92 double breadth_explore(int root, int o, int pprint)
93 {
94     unsigned long current;
95     struct signature * s;
96     int depth = 0;
97     double dist = 0;
98     queue q;
99
100     do_all(&breadth_explore_init);
101
102     colors[root] = GRAY;
103     depths[root] = 0;
104
105     queue_reset(&q);
106     enqueue(root, &q);
107
108     while(!queue_empty(&q))
109     {
110         current = dequeue(&q);
111
112         if (o == BACKWARDS)
113             s = edges[current].head;
114         else
115             s = redges[current].head;
116
117         while (s)
118         {
119             if (colors[next(s->s,o)] == WHITE)
120             {
121                 if (depth == depths[current])
122                     depth++;
123
124                 colors[next(s->s,o)] = GRAY;
125                 depths[next(s->s,o)] = depths[current]+1;
126
127                 if (is_tmark(next(s->s, o)))
128                     dist+=depth;
129
130                 enqueue(next(s->s, o), &q);
131             }
132             s = s->next;
133         }
134         colors[current] = BLACK;
135     }
136
137     return dist;
138 }
```

Listing 5 – marks.c : Marquage de clés

```

1  #include "main.h"
2
3
4  extern key vertices[MAXKEYS];
5
6  extern struct sig sigs[MAXTSIGS];
7  extern unsigned int num_sigs;
8  extern unsigned long num_keys;
9
10 extern char colors[MAXKEYS];
11 extern char depths[MAXKEYS];
12
13 extern signatures edges[MAXKEYS]; // for BACKWARDS: list of signers for each signee
14 extern signatures redges[MAXKEYS]; // for FORWARDS: list of signees for each signer
15
16 extern int select_mode;
17
18 extern double get_kx(unsigned long pos);
19 extern double get_ky(unsigned long pos);
20 extern unsigned long get_pos_from_id(unsigned long id);
21
22
23 void setm(unsigned long pos, char val) { vertices[pos].mark = val; }
24 void delm(unsigned long pos, char val) { vertices[pos].mark &= ~val; }
25 void rstm(unsigned long pos) { delm(pos, MALL); }
26 void addm(unsigned long pos, char val) { vertices[pos].mark |= val; }
27 void notm(unsigned long pos, char val) { vertices[pos].mark ^= val; }
28 char getm(unsigned long pos) { return vertices[pos].mark; }
29 char hasm(unsigned long pos, char val) { return (getm(pos) & val) == val; }
30
31 char findm(char val)
32 {
33     unsigned long i;
34
35     for (i=0; i<num_keys; i++)
36         if (hasm(i, val))
37             return i;
38
39     return num_keys;
40 }
41
42 char is_in_tld(unsigned long pos, char* tld)
43 {
44     return (!strcascmp(vertices[pos].name + (strlen(vertices[pos].name) -
45         strlen(tld))*sizeof(char), tld));
46 }
47
48 unsigned long count_tld(char* tld)
49 {
50     unsigned int i, tot=0;
51     for (i=0; i<num_keys; i++)
52         if (is_in_tld(i, tld))
53             tot++;
54     return tot;
55 }
56
57 void addm_all(char val) { do_all_c(&addm, val); }
58 void delm_all(char val) { do_all_c(&delm, val); }
59 void notm_all(char val) { do_all_c(&notm, val); }
60 void setm_all(char val) { do_all_c(&setm, val); }
61 void delm_selected(char val) { oprm_all(val, MU, SELECT.REMOVE); }
62 void addm_selected(char val) { oprm_all(val, MU, SELECT.UNION); }
63
64 void addm_rand(unsigned long num, char val)
65 {
66     // Add mark val to num random keys.
67     // TODO should use malloc
68
69     unsigned long s[MAXKEYS], i, size=num_keys, buf, chosen;
70
71     // initialise identity permutation
72     for (i=0; i<size; i++)
73         s[i]=i;
74
75     for (i=0; i<num; i++)
76     {
77         // mark a key, and put it at the end of the permutation so that it will not get marked again
78         chosen=(unsigned long) ((double) size * ((double) rand() * (double) rand()) / ((RANDMAX +
79         1.0)*(RANDMAX + 1.0)));
80         buf = s[chosen]; s[chosen] = s[size]; s[size] = buf;
81         addm(s[size], val);
82     }
83 }

```

```

80     size--;
81 }
82 }
83
84
85 void mark(unsigned long pos) { addm(pos, MFT); }
86 void fmark(unsigned long pos) { addm(pos, MF); }
87 void tmark(unsigned long pos) { addm(pos, MT); }
88 void unmark(unsigned long pos) { delm(pos, MFT); }
89 void mark_all() { addm_all(MFT); }
90 void unmark_all() { delm_all(MFT); }
91 void mark_rand(unsigned long num) { addm_rand(num, MFT); }
92 void fmark_rand(unsigned long num) { addm_rand(num, MF); }
93 void tmark_rand(unsigned long num) { addm_rand(num, MT); }
94
95 char is_fmark(unsigned long pos) { return hasm(pos, MF); }
96 char is_tmark(unsigned long pos) { return hasm(pos, MT); }
97 char is_mark(unsigned long pos) { return hasm(pos, MFT); }
98 char is_seen(unsigned long pos) { return hasm(pos, MS); }
99
100 unsigned long count_m(char val)
101 {
102     // Count val marks.
103     unsigned long i, tot=0;
104     for(i=0; i<num_keys; i++)
105         if (hasm(i, val))
106             tot++;
107     return tot;
108 }
109
110 unsigned long count_marks() { return count_m(MFT); }
111 unsigned long count_fmmarks() { return count_m(MF); }
112 unsigned long count_tmmarks() { return count_m(MT); }
113 int selection_empty() { return (count_m(MU) == 0); }
114
115 void addm_neighbours(unsigned long pos, char m, char direction)
116 {
117     struct signature * s;
118
119     if (direction == FORWARDS) s = redges[pos].head;
120     else s = edges[pos].head;
121
122     while (s)
123     {
124         if (direction == FORWARDS)
125             addm(s->s->t, m);
126         else
127             addm(s->s->f, m);
128         s = s->next;
129     }
130 }
131
132
133 void do_by_tld(void (*fun)(unsigned int, char), char* tld, char val)
134 {
135     unsigned int i;
136     for (i=0; i<num_keys; i++)
137         if (is_in_tld(i, tld))
138             (*fun)(i, val);
139 }
140
141 void add_by_tld(char* tld, char val) { do_by_tld(&addm, tld, val); }
142 void del_by_tld(char* tld, char val) { do_by_tld(&delm, tld, val); }
143
144 void mark_by_tld(char* tld) { add_by_tld(tld, MFT); }
145 void fmark_by_tld(char* tld) { add_by_tld(tld, MF); }
146 void tmark_by_tld(char* tld) { add_by_tld(tld, MT); }
147 void unmark_by_tld(char* tld) { del_by_tld(tld, MFT); }
148
149
150 void oprm(unsigned long i, char a, char b, char op)
151 {
152     // Applies operator to marks a and b.
153
154     switch(op)
155     {
156     case SELECT_SET:
157         if (hasm(i, b)) addm(i, a); else delm(i, a);
158         break;
159     case SELECT_UNION:
160         if (hasm(i, b)) addm(i, a);
161         break;
162     case SELECT_INTERSECTION:
163         if (hasm(i, b) && hasm(i, a)) addm(i, a); else delm(i, a);

```

```
164         break;
165     case SELECT.REMOVE:
166         if (hasm(i, b)) delm(i, a);
167         break;
168     case SELECT.XOR:
169         if (((hasm(i, b)) || hasm(i, a)) && !(hasm(i, b) && hasm(i, a))) addm(i, a); else delm(i,
170             a);
171         break;
172     }
173 }
174 void oprm_all(char a, char b, char op)
175 {
176     unsigned long i;
177     for (i=0; i<num_keys; i++)
178         oprm(i, a, b, op);
179 }
180
181 void opsel_by_tld(char* tld)
182 {
183     add_by_tld(tld, MV);
184     printf("Marked keys ending in %s\n", tld);
185     oprm_all(MU, MV, select_mode);
186     delm_all(MV);
187 }
188
189 void opsel_by_id_p(unsigned long id_p)
190 {
191     addm(get_pos_from_id(id_p), MV);
192     printf("Marked key %lx\n", id_p);
193     oprm_all(MU, MV, select_mode);
194     delm_all(MV);
195 }
196
197 void opsel_by_id(char* id) { opsel_by_id_p(strtol(id, NULL, 16)); }
198
199 void opsel_by_region_p(double x1, double y1, double x2, double y2)
200 {
201     unsigned long i;
202     double kx, ky;
203
204     for (i=0; i<num_keys; i++)
205     {
206         kx = get_kx(i); ky = get_ky(i);
207         if (((x1 < kx && kx < x2) || ((x2 < kx) && (kx < x1)))
208             && ((y1 < ky && ky < y2) || ((y2 < ky) && (ky < y1))))
209             addm(i, MV);
210     }
211
212     oprm_all(MU, MV, select_mode);
213     delm_all(MV);
214
215     redraw_all(0);
216 }
217
218 void opsel_rand(char* num)
219 {
220     unsigned long num_p;
221
222     num_p = strtol(num, NULL, 10);
223     addm_rand(num_p, MV);
224     printf("Marked %lu random keys.\n", num_p);
225     oprm_all(MU, MV, select_mode);
226     delm_all(MV);
227 }
228
229 void addm_m_neighbours(char m1, char m2, enum graph_orientation o)
230 {
231     // Add mark m2 to neighbours of keys with m1 (neighbour means signer or signee according to the
232     // orientation).
233
234     unsigned long i;
235
236     for (i=0; i<num_keys; i++)
237         if (hasm(i, m1))
238             addm_neighbours(i, m2, o);
239 }
240
```


Listing 6 – distances.c : Calculs de distances

```

1  #include "main.h"
2
3
4  extern unsigned long num_keys;
5  extern double breadth_explore(int root, int o, int pprint);
6
7  extern double get_kx(unsigned long pos);
8  extern double get_ky(unsigned long pos);
9
10 char is_seen(unsigned long pos);
11
12
13
14 double get_dists(unsigned long pos)
15 {
16     return breadth_explore(pos, BACKWARDS, 0);
17 }
18
19
20 double get_gdists(unsigned long pos)
21 {
22     unsigned long j;
23     double total_dist=0;
24     double dx, dy;
25
26     for (j=0; j<num_keys; j++)
27         if(is_tmark(j))
28         {
29             // add distance between i and j
30             dx = get_kx(pos) - get_kx(j);
31             dy = get_ky(pos) - get_ky(j);
32             total_dist+= sqrt(dx*dx+dy*dy);
33         }
34
35     return total_dist;
36 }
37
38
39 double calc_dists(char printstep, unsigned long n_keys, unsigned long brk, float pb_f, float
    pb_t, double (*get)(unsigned long))
40 {
41     // Sums all distances between FROM keys and TO keys.
42     // n_keys is passed so that marked nodes don't need to be counted again.
43     // brk is used to approximate the result with only the brk first keys (obsolete).
44     // pb_f and pb_t indicate the minimal and maximal progress bar values
45
46     unsigned long i, ok_keys=0;
47     double total_dist=0;
48
49     for (i=0; i<num_keys; i++)
50         if(is_fmark(i))
51         {
52             // count all distances from this key
53             total_dist+=(*get)(i);
54             ok_keys++;
55
56             if (printstep && !(ok_keys % printstep))
57             {
58                 // display status every now and then, and update progress bar
59                 printf("Checked %lu keys, total dist is %lf, progress is %lf between %lf and %lf.\n", i,
60                     total_dist, pb_f + ( ( (double) ok_keys)/n_keys )*(pb_t - pb_f), pb_f, pb_t);
61                 if (progress_bar((pb_f + ( ( (double) ok_keys)/n_keys )*(pb_t - pb_f))))
62                     return -1.0;
63             }
64             if (brk && ok_keys>brk) break;
65         }
66     if (brk && ok_keys>brk)
67         return (total_dist*n_keys)/ok_keys; // estimate
68     else return total_dist; // exact result
69 }
70
71 double total_dists(char printstep, unsigned long n_keys, unsigned long brk, float pb_f, float
    pb_t)
72 {
73     // Sums all graph distances between FROM keys and TO keys.
74     return calc_dists(printstep, n_keys, brk, pb_f, pb_t, &get_dists);
75 }
76
77
78 double total_gdists(char printstep, unsigned long n_keys, unsigned long brk, float pb_f, float

```

```
79     pb_t)
80 {
81     // Sums all distances between FROM keys and TO keys in graphical representation.
82     return calc_dists(printstep, n_keys, brk, pb_f, pb_t, &get_gdists);
83 }
84
85 double dist_between_tlds(char *tld1, char* tld2, char precision, float pb_f, float pb_t)
86 {
87     // Computes total distance between two TLDs.
88     unmark_all();
89     fmark_by_tld(tld1); tmark_by_tld(tld2);
90
91     unsigned long n_fkeys=count_fmmarks(), n_tkeys=count_tmmarks();
92
93     printf("For the %lu keys in TLD1 %s to the %lu keys in TLD2 %s: computing...\n", n_fkeys, tld1,
94           n_tkeys, tld2);
95
96     double dist = total_dists(DISPRATE, n_fkeys, precision, pb_f, pb_t);
97
98     printf("For the %lu keys in TLD1 %s to the %lu keys in TLD2 %s: %lf (precision %d)\n", n_fkeys,
99           tld1, n_tkeys, tld2, dist/(n_fkeys*n_tkeys), precision);
100
101     unmark_all();
102
103     return dist;
104 }
105
106
107 double fdist_between_tld_and_rand(char *tld, unsigned long num, char precision, float pb_f, float
108     pb_t)
109 {
110     // Computes distance from TLD to random set.
111     unmark_all();
112     fmark_by_tld(tld); tmark_rand(num);
113
114     unsigned long n_fkeys=count_fmmarks(), n_tkeys=count_tmmarks();
115
116     double dist = total_dists(DISPRATE, n_fkeys, precision, pb_f, pb_t);
117
118     printf("For the %lu keys in TLD1 %s TO %lu random keys: %lf (precision %d)\n", n_fkeys, tld,
119           n_tkeys, dist/(n_fkeys*n_tkeys), precision);
120
121     unmark_all();
122
123     return dist;
124 }
125
126
127 double tdist_between_tld_and_rand(char *tld, unsigned long num, char precision, float pb_f, float
128     pb_t)
129 {
130     // Computes distance from random set to TLD.
131     unmark_all();
132     tmark_by_tld(tld); fmark_rand(num);
133
134     unsigned long n_fkeys=count_fmmarks(), n_tkeys=count_tmmarks();
135
136     double dist = total_dists(DISPRATE, n_fkeys, precision, pb_f, pb_t);
137
138     printf("For the %lu keys in TLD1 %s FROM %lu random keys: %lf (precision %d)\n", n_tkeys, tld,
139           n_fkeys, dist/(n_fkeys*n_tkeys), precision);
140
141     unmark_all();
142
143     return dist;
144 }
145
146
147 double dist_between_rand(unsigned long num1, unsigned long num2, char precision, float pb_f,
148     float pb_t)
149 {
150     // Computes distance between random sets.
151     unmark_all();
152     fmark_rand(num1); tmark_rand(num2);
153
154 }
```

```
155     double dist = total_dists(DISPRATE,num1,precision , pb_f, pb_t);
156
157     printf("For %lu random keys to %lu random keys: %lf (precision %d)\n", num1,
158           num2,dist/(num1*num2), precision);
159
160     unmark_all();
161     return dist;
162 }
163
164
165
166 void two_tlds_vs_rand(char *tld1, char*tld2, char precision)
167 {
168     // Compares two TLDs.
169
170     unsigned long n_fkeys=count_tld(tld1), n_tkeys=count_tld(tld2);
171
172     dist_between_tlds(tld1, tld2, precision, 0, 1);
173     fdist_between_tld_and_rand(tld1, n_tkeys, precision, 0, 1);
174     tdist_between_tld_and_rand(tld2, n_fkeys, precision, 0, 1);
175     dist_between_rand(n_fkeys, n_tkeys, precision, 0, 1);
176 }
177
178
179 double dist_of_tld(char* tld, char precision, float pb_f, float pb_t)
180 {
181     // Computes distance within a TLD.
182
183     return dist_between_tlds(tld, tld, precision, pb_f, pb_t);
184 }
185
186
187 double dist_of_rand(unsigned long num, char precision, float pb_f, float pb_t)
188 {
189     // Computes distance within a random set.
190     // This is not the same thing as dist_between_rand(x, x, ...) which would compare two different
191     // random sets of the same size.
192
193     unmark_all();
194     mark_rand(num);
195
196     double dist = total_dists(DISPRATE,num,precision , pb_f, pb_t);
197
198     printf("For %lu random keys: %lf (precision %d)\n", num, dist/(num*num), precision);
199
200     unmark_all();
201
202     return dist;
203 }
204
205
206 int all_tlds_vs_rand(char precision, FILE* out)
207 {
208     // Compares all TLDs to random sets, and outputs the results to a file.
209
210     unsigned long i, num, seen=0;
211     double dist1, dist1g, dist2, dist2g, alldist, alldistg;
212     char tld[10], str[2000];
213
214     // MS is used to mark keys from TLDs we have already seen
215     delm_all(MS);
216
217     // headers
218     if (out) fprintf(out, "TLD;Nombre;Dist. TLD;Dist. graph. TLD;Dist. rand.;Dist. graph.
219         rand;Diff.;Diff. graph.\n");
220
221     for (i=0;i<num_keys;i++)
222     {
223         if(!is_seen(i))
224         {
225             tld[0] = '\0';
226             get_tld(i, &tld, 8);
227             if(tld[0] == '.') // ignore garbage TLDs
228             {
229                 // examine this TLD
230                 printf("TLD %s, index %lu\n", tld, i);
231
232                 add_by_tld(&tld,MS);
233
234                 num = count_tld(tld);
235
236                 // distance within TLD
```

```

236     dist1 = dist_of_tld((char*) &tld, precision, 0.25*((float) seen / (float) (num_keys+1)),
237         0.25*((float) seen / (float) (num_keys+1)) + .5*((float) num / (2 * (float)
238         (num_keys+1))) );
239     if (dist1 < 0) // user cancel
240         return -1;
241     unmark_all();
242     mark_by_tld(tld);
243     dist1g = total_gdists(DISPRATE, num, precision, 0.25*((float) seen / (float)
244         (num_keys+1)) + 0.5*((float) num / (2 * (float) (num_keys+1))), 0.25*((float) seen /
245         (float) (num_keys+1)) + ((float) num / (2 * (float) (num_keys+1))) );
246     unmark_all();
247     if (dist1g < 0) // user cancel
248         return -1;
249     // distance within random set
250     dist2 = dist_of_rand(num, precision, 0.25*((float) seen / (float) (num_keys+1)) +
251         ((float) num / (2 * (float) (num_keys+1))), 0.25*((float) seen / (float)
252         (num_keys+1)) + 0.75*((float) num / ((float) (num_keys+1))) );
253     if (dist2 < 0) // user cancel
254         return -1;
255     mark_rand(num);
256     dist2g = total_gdists(DISPRATE, num, precision, 0.25*((float) seen / (float)
257         (num_keys+1)) + 0.75*((float) num / ((float) (num_keys+1))), 0.25*((float) seen /
258         (float) (num_keys+1)) + ((float) num / ((float) (num_keys+1))) );
259     unmark_all();
260     if (dist2g < 0) // user cancel
261         return -1;
262     strip_gt(&tld);
263     sprintf(str, "%s;%ld;%lf;%lf;%lf;%lf;%lf;%lf\n", tld, num, dist1/((double) num*num),
264         dist1g/((double) num*num), dist2/((double) num*num), dist2g/((double) num*num),
265         (dist2-dist1)/((double) num*num), (dist2g-dist1g)/((double) num*num));
266     printf("%s", str); if (out) fprintf(out, "%s", str);
267     seen += num;
268 }
269 }
270 // grand total
271 mark_all();
272 alldist = total_dists(DISPRATE, num_keys, 0, .5, .75);
273 alldistg = total_gdists(DISPRATE, num_keys, 0, .75, 1);
274 sprintf(str, "%s;%ld;%lf;%lf;%lf;%lf;%lf;%lf\n", "TOTAL", num_keys, alldist/((double)
275     num_keys*num_keys), alldistg/((double) num_keys*num_keys), alldist/((double)
276     num_keys*num_keys), alldistg/((double) num_keys*num_keys), 0, 0);
277 printf("%s", str); if (out) fprintf(out, "%s", str);
278 unmark_all();
279 return 0;
280 }
281
282 unsigned long all_tlds_header(FILE* out, unsigned long min)
283 {
284     // Create header.
285     unsigned long i, num, total_keys=0;
286     char tld[10];
287     delm_all(MS);
288     printf(";", tld); if (out) fprintf(out, ":", tld);
289     for (i=0; i<num_keys; i++)
290     {
291         if(!is_seen(i))
292         {
293             tld[0] = '\0';
294             get_tld(i, &tld, 8);
295             if(tld[0] == '.')
296             {
297                 add_by_tld(&tld, MS);
298                 num = count_tld(tld);
299                 if (num >= min) // ignore small TLDS
300                 {
301                     printf("header: TLD %s, index %lu, num %lu\n", tld, i, num);
302                     strip_gt(&tld);
303                     printf(";%s\n", tld); if (out) fprintf(out, ":%s", tld);
304                 }
305             }
306         }
307     }

```

```

308         total_keys += num;
309     }
310 }
311 }
312 }
313
314 // end of header
315 printf("\n"); if (out) fprintf(out, "\n");
316
317 delm_all(MS);
318
319 return total_keys;
320 }
321
322
323
324 int all_tlds_to_all_tlds(char precision, FILE* out, unsigned long min)
325 {
326     // Compares all TLDs to each other.
327
328     unsigned long i, j, num, num2, seen=0, seen2=0, total_keys=0;
329     double dist;
330     char tld[10], tlds[10], tld2[10];
331
332     total_keys = all_tlds_header(out, min);
333
334     // MS and MU are used
335     delm_all(MS);
336
337     for (i=0; i<num_keys; i++)
338     {
339         if(!is_seen(i))
340         {
341             delm_all(MU);
342             tld[0] = '\0';
343             get_tld(i, &tld, 8);
344             if(tld[0] == '.')
345             {
346                 num = count_tld(tld);
347                 add_by_tld(&tld, MS);
348                 if (num >= min) // ignore small TLDs
349                 {
350                     printf("TLD %s, index %lu\n", tld, i);
351                     strcpy(tlds, tld);
352                     strip_gt(tlds);
353                     if (out) fprintf(out, "%s", tlds);
354                     seen2=0;
355
356                     for (j=0; j<num_keys; j++)
357                     {
358                         if(!hasm(j, MU))
359                         {
360                             tld2[0] = '\0';
361                             get_tld(j, &tld2, 8);
362                             if(tld2[0] == '.')
363                             {
364                                 num2 = count_tld(tld2);
365                                 add_by_tld(&tld2, MU);
366                                 if (num2 >= min) // ignore small TLDs
367                                 {
368                                     // compare TLDs tld and tld2
369
370                                     printf("- TLD %s, index %lu\n", tld2, j);
371
372                                     // TODO progress bar is wrong
373                                     dist = dist_between_tlds((char*) &tld, (char*) &tld2, precision, (((float)
374                                         seen) / total_keys) + (((float) num * seen2) / (total_keys * total_keys)),
375                                         (((float) seen) / total_keys) + (((float) num * (seen2 + num2)) /
376                                         (total_keys * total_keys)));
377                                     if (dist < 0) // user cancel
378                                         return -1;
379
380                                     strip_gt(&tld2);
381                                     printf("from %s to %s: %lf total, %lf average\n", tld, tld2, dist,
382                                         dist/((double) num*num2));
383                                     if (out) fprintf(out, ":%.2lf", tld, dist/((double) num*num2));
384                                     seen2 += num;
385                                 }
386                             }
387                         }
388                     }
389                 }
390             }
391         }
392     }
393
394     if (out) fprintf(out, ":%s", tlds);
395     seen += num;

```

```
388
389     printf("\n"); if (out) fprintf(out, "\n");
390 }
391 }
392 }
393 }
394
395 delm_all(MU);
396
397 all_tlds_header(out, min);
398
399 return 0;
400 }
```

Listing 7 – force.c : Algorithme force-directed

```

1  #include "main.h"
2
3
4  char colors[MAXKEYS];
5  char depths[MAXKEYS];
6
7  unsigned long gqueue[MAXSIGs];
8  int q_head;
9  int q_tail;
10
11 signatures edges[MAXKEYS]; // for BACKWARDS: list of signers for each signee
12 signatures redges[MAXKEYS]; // for FORWARDS: list of signees for each signer
13
14 extern double delta_x, delta_y;
15
16 extern int moving;
17
18 extern key vertices[MAXKEYS];
19
20 extern struct sig sigs[MAXTSIGs];
21 extern unsigned int num_sigs;
22 extern unsigned long num_keys;
23
24
25 void set_key_loc_o(unsigned long pos, unsigned char comp, double val)
26 {
27     // override check
28     vertices[pos].p[comp] = val;
29 }
30 void set_key_loc(unsigned long pos, unsigned char comp, double val)
31 {
32     // check first that the key isn't being moved by the user
33     if (!(moving && hasm(pos, MU)))
34         set_key_loc_o(pos, comp, val);
35 }
36 double get_key_loc(unsigned long pos, unsigned char comp) { return vertices[pos].p[comp]; }
37
38 void set_kx(unsigned long pos, double val) { set_key_loc(pos, X, val); }
39 void set_ky(unsigned long pos, double val) { set_key_loc(pos, Y, val); }
40 void set_kx_o(unsigned long pos, double val) { set_key_loc_o(pos, X, val); }
41 void set_ky_o(unsigned long pos, double val) { set_key_loc_o(pos, Y, val); }
42 double get_kx(unsigned long pos) { return get_key_loc(pos, X); }
43 double get_ky(unsigned long pos) { return get_key_loc(pos, Y); }
44
45 void set_key_accel(unsigned long pos, unsigned char comp, double val) { vertices[pos].a[comp] =
    val; }
46 double get_key_accel(unsigned long pos, unsigned char comp) { return vertices[pos].a[comp]; }
47
48 void set_kax(unsigned long pos, double val) { set_key_accel(pos, X, val); }
49 void set_kay(unsigned long pos, double val) { set_key_accel(pos, Y, val); }
50 double get_kax(unsigned long pos) { return get_key_accel(pos, X); }
51 double get_kay(unsigned long pos) { return get_key_accel(pos, Y); }
52
53
54 double update_ep(unsigned long sig, char update)
55 {
56     // Updates signature potential energy or returns it.
57
58     if (update)
59     {
60         double dx=(vertices[sigs[sig].f].p[X] - vertices[sigs[sig].t].p[X]);
61         double dy=(vertices[sigs[sig].f].p[Y] - vertices[sigs[sig].t].p[Y]);
62         double x = sqrt(dx*dx + dy*dy)-L0;
63         sigs[sig].ep = 0.5 * K1 * x*x;
64     }
65     return sigs[sig].ep;
66 }
67
68
69 double update_eps(char update)
70 {
71     // Updates potential energy for all keys and returns total.
72
73     unsigned long i;
74     double tot=0;
75
76     for (i=0; i<num_sigs; i++)
77         tot+=update_ep(i, update);
78
79     return tot;
80 }

```

```

81
82
83 double update_key_eps(unsigned long pos, char update)
84 {
85     // Updates potential energy for all signatures of a given key.
86
87     double tot=0;
88     struct signature* s=redges[pos].head;
89
90     while (s)
91     {
92         tot += update_ep(s->s->id, update);
93         s = s->next;
94     }
95
96     // TODO count signatures in both directions
97
98     /* s=edges[pos].head;
99     while (s)
100     {
101         tot += update_ep(s->s->id, update);
102         s = s->next;
103     }*/
104
105     /* electrostatic potential energy (disabled for performance reasons)
106
107     for (i=0; i<num_keys; i++)
108     {
109         dx = vertices[pos].a[X] - vertices[i].a[X];
110         dy = vertices[pos].a[Y] - vertices[i].a[Y];
111         if (dx && dy)
112             tot+=E/sqrt(dx*dx + dy*dy);
113     }*/
114
115     return tot;
116 }
117
118
119 void key_alter_accel(unsigned long pos, struct signature* s, char o)
120 {
121     // Update key acceleration to take a signature into account.
122
123     double l,dx,dy,px,py;
124
125     dx = (vertices[pos].p[X] - vertices[next(s->s, o)].p[X]);
126     dy = (vertices[pos].p[Y] - vertices[next(s->s, o)].p[Y]);
127     l = sqrt(dx*dx + dy*dy); // l is the distance between the keys
128
129     // (px, py) is the equilibrium position of spring
130     px = vertices[next(s->s, o)].p[X]+ dx*L0 / l;
131     py = vertices[next(s->s, o)].p[Y]+ dy*L0 / l;
132
133     // add this to the acceleration (applying Hooke's law)
134     vertices[pos].a[X] += K2 * (px-vertices[pos].p[X]);
135     vertices[pos].a[Y] += K2 * (py-vertices[pos].p[Y]);
136 }
137
138 void report_key_loc(unsigned long pos) { printf("Key %lu is at %lf %lf\n", pos, get_kx(pos),
139     get_ky(pos)); }
140 void key_loc(unsigned long pos)
141 {
142     // Display key name and position.
143     name_key_p(pos); report_key_loc(pos);
144 }
145
146 void update_key_accel(unsigned long pos)
147 {
148     // Update key acceleration.
149
150     struct signature* s=redges[pos].head;
151
152     // reset acceleration
153     vertices[pos].a[X] = 0;
154     vertices[pos].a[Y] = 0;
155
156     while (s)
157     {
158         // take all signatures into account
159         key_alter_accel(pos, s, FORWARDS);
160         s = s->next;
161     }
162
163     //TODO bidirectional

```



```

164  /*
165  s = edges[pos].head;
166
167  while (s)
168  {
169      key_alter_accel(pos, s, BACKWARDS);
170      s = s->next;
171  }*/
172
173  /* electrostatic (disabled)
174  for (i=0; i<num_keys; i++)
175  {
176      dx = vertices[pos].a[X] - vertices[i].a[X];
177      dy = vertices[pos].a[Y] - vertices[i].a[Y];
178      if (dx && dy)
179      {
180          set_kax(pos, get_kax(pos) + (E / (dx * dx)));
181          set_kay(pos, get_kay(pos) + (E / (dy * dy)));
182      }
183  }*/
184  }
185
186
187  void move_key(unsigned long pos, double step)
188  {
189      // Move key following its acceleration.
190      // The algorithm tries to find a position on the ray which minimises potential energy.
191
192      double ep1, ep2, x, y, delta;
193
194      ep1 = update_key_eps(pos, 0);
195      ep2 = ep1;
196      delta = 1; // for the loop
197
198      // we move along the ray with bigger and bigger steps, as long as potential energy decreases
199      // when it starts increasing again, we revert back to the last position
200      // TODO improve (dichotomy)
201
202      while (delta > 0)
203      {
204          // we move, and update ep
205          ep1 = ep2;
206          x = get_kx(pos);
207          y = get_ky(pos);
208
209          set_kx(pos, x + step*get_kax(pos));
210          set_ky(pos, y + step*get_kay(pos));
211
212          ep2 = update_key_eps(pos, 1);
213          delta = ep1 - ep2;
214          step*=MULT.STEP;
215      }
216
217      // rollback
218      set_kx(pos, x);
219      set_ky(pos, y);
220      ep1 = update_key_eps(pos, 1);
221  }
222
223
224  void move_one_pass(double step, unsigned long number)
225  {
226      unsigned long s[MAXKEYS]; //use malloc
227      unsigned long i;
228      unsigned long size=num_keys;
229      unsigned long buf,chosen;
230
231      // initialise identity permutation
232      for (i=0; i<size; i++)
233          s[i]=i;
234
235      for (i=0; i<(number>num_keys?num_keys:number); i++)
236      {
237          // mark a key, and put it at the end of the permutation
238          // so that it will not get marked again
239          chosen=(unsigned long) (((double) size * ((double) rand() * (double) rand()) / ((RAND.MAX +
240              1.0)*(RAND.MAX + 1.0))));
241          buf = s[chosen];
242          s[chosen] = s[size];
243          s[size] = buf;
244          {
245              update_key_accel(buf);
246              move_key(buf, step);
247          }
248      }

```

```
247     size --;
248 }
249
250 }
251
252
253 void move_key_step(unsigned long i)
254 {
255     move_key(i, STEP);
256 }
257
258
259 void moving_adjust(unsigned long pos)
260 {
261     // isn't there a way to avoid globals?
262
263     set_kx_o(pos, get_kx(pos) + delta_x);
264     set_ky_o(pos, get_ky(pos) + delta_y);
265 }
```

Listing 8 – events.c : Gestion des événements

```

1  #include "main.h"
2
3  // for rectangle selection
4  #define RECT_MAYBE 2
5  #define RECT_YES 1
6  #define RECT_NO 0
7
8
9  extern double ax, ay, bx, by; // frame
10
11  char inputting=0; // is user input currently taking place?
12  char moving=0; // is user currently moving keys around?
13  double delta_x, delta_y;
14
15  extern key vertices[MAXKEYS];
16
17  extern char tooltip;
18  extern char tooltip_time;
19  extern void *input_callback;
20
21  // when displaying info about a key which is close to the mouse pointer, temporarily store some
   information about this key
22  unsigned long last_key = 0;
23  char had_mn=0;
24  char had_mi=0;
25  SDL_Color had_color;
26
27  extern int width;
28  extern int height;
29
30  extern int auto_recalc;
31
32  // for rectangle selection
33  extern int mouse_x, mouse_y;
34  // to display rectangle on mouse move
35  extern char mouse_byrect;
36
37  // for mouse button drag and drop
38  extern int mouse_mx, mouse_my;
39  // matching real coordinates
40  extern double mouse_mx_r, mouse_my_r;
41  // to refresh on mouse move
42  extern char mouse_middle;
43
44  // Always contains up-to-date mouse coordinates.
45  int pointer_x, pointer_y;
46
47  extern char status[500];
48
49  extern unsigned long num_keys;
50
51  extern TTF_Font *font;
52  extern TTF_Font *small_font;
53
54  extern SDL_Surface *screen;
55  extern SDL_InputTTF *ttf;
56  extern SDL_Rect inputpos;
57
58  extern int select_mode;
59
60  extern double disp2real_x(double disp_x);
61  extern double disp2real_y(double disp_y);
62  extern unsigned long find_closest_m(double x, double y, int val, double dist);
63  extern unsigned long find_closest(double x, double y, double dist);
64  extern void opsel_by_id_p(unsigned long id_p);
65  extern void opsel_by_region_p(double x1, double y1, double x2, double y2);
66
67  extern double total_dists(char printstep, unsigned long n_keys, unsigned long brk, float pb_f,
   float pb_t);
68  extern double total_gdists(char printstep, unsigned long n_keys, unsigned long brk, float pb_f,
   float pb_t);
69  extern void in_circle(char* radius);
70
71
72
73  int all_tlds_to_all_tlds_w(char* filename)
74  {
75      // Callback function for all_tlds_to_all_tlds
76
77      FILE* f;
78      int rsl;

```

```
79
80     f = fopen(filename , "w");
81     if (!f)
82     {
83         help("Could not open file!\n");
84         return 1;
85     }
86
87     help("Computing all_tlds_to_all_tlds...\n");
88
89     rsl = all_tlds_to_all_tlds(0, f, 300);
90
91     fclose(f);
92
93     if (rsl == -1) help("Results of all_tlds_to_all_tlds saved.\n");
94     else help("User abort.\n");
95
96     return rsl;
97 }
98
99
100 int all_tlds_vs_rand_w(char* filename)
101 {
102     // Callback function for all_tlds_vs_rand
103
104     FILE* f;
105     int rsl;
106
107     f = fopen(filename , "w");
108     if (!f)
109     {
110         help("Could not open file!\n");
111         return 1;
112     }
113
114     help("Computing all_tlds_vs_rand...\n");
115
116     rsl = all_tlds_vs_rand(0, f);
117
118     fclose(f);
119
120     if (rsl == -1) help("Results of all_tlds_vs_rand saved.\n");
121     else help("User abort.\n");
122
123     return rsl;
124 }
125
126
127 int manage_event()
128 {
129     // Manage next SDL event.
130
131     char msg[500];
132     unsigned long nfrom, nto;
133     double total-dist, total-gdist;
134
135     int dx, dy;
136     float ddx, ddy;
137     unsigned long key;
138
139     unsigned long color;
140
141     SDL_Event event;
142
143     if (!SDL_PollEvent(&event)) return 0; // no more events
144
145     switch(event.type)
146     {
147     case SDLKEYDOWN:
148         if (inputting)
149         {
150             // an input field is currently displayed
151
152             SDL_Input_TTF_Input( ttf, screen, &event );
153             switch (event.key.keysym.sym)
154             {
155             case SDLK_RETURN:
156                 // send input to callback function
157                 printf("> %s\n", ttf->input->string);
158                 (*((void (*)(char*)) input_callback))(ttf->input->string);
159                 stop_input();
160                 break;
161
162             case SDLK_ESCAPE:
```

```

163         // cancel input
164         printf("> [ESC]\n");
165         stop_input();
166         break;
167     }
168     redraw();
169
170 } else {
171
172     if (SDL_GetModState() & KMOD_CTRL)
173     {
174         switch (event.key.keysym.sym)
175         {
176             case SDLK_a:
177                 user_input("Please enter a file name to save the results of all_tlds-vs-rand:",
178                             &all_tlds_vs_rand_w, "all_tlds-vs-rand.csv", strlen("all_tlds-vs-rand"));
179                 break;
180             case SDLK_b:
181                 user_input("Please enter a file name to save the results of all_tlds-to-all_tlds:",
182                             &all_tlds_to_all_tlds_w, "all_tlds-to-all_tlds.csv",
183                             strlen("all_tlds-to-all_tlds"));
184                 break;
185             case SDLK_c:
186                 printf("%d\n", event.button.x);
187                 user_input("Please enter the circle radius in pixels:", &in_circle, "", 0);
188         }
189     }
190     else if (SDL_GetModState() & KMOD_SHIFT)
191     {
192         switch (event.key.keysym.sym)
193         {
194             case SDLK_a:
195                 delm_all(MU); help("All keys unselected.");
196                 break;
197             case SDLK_c:
198                 // TODO: intelligent autocolour
199                 color = rand() % (1 << 24);
200                 sprintf(msg, "0x%x", color);
201                 opsel_by_color(msg);
202                 sprintf(msg, "Marked with random color 0x%x.", color);
203                 help(msg);
204                 break;
205             case SDLK_d:
206                 delm_all(MFT); help("From and To marks removed.");
207                 break;
208             case SDLK_e: help("TODO: Select by name."); break;
209             case SDLK_f:
210                 addm_selected(MT); help("Keys tagged as To.");
211                 break;
212             case SDLK_g:
213                 delm_selected(MA); help("Key accel hidden for selected keys.");
214                 break;
215             case SDLK_i:
216                 delm_selected(MI); help("Key IDs hidden for selected keys.");
217                 break;
218             case SDLK_k: help("TODO: Select by color."); break;
219             case SDLK_l:
220                 do_all(&update_key_accel);
221                 break;
222             case SDLK_m:
223                 do_all(&move_key_step);
224                 break;
225             case SDLK_n:
226                 delm_selected(MN); help("Keys names hidden for selected keys.");
227                 break;
228             case SDLK_r: help("TODO: Intelligent random keys (excluding already selected if
229                             adding, etc.)"); break;
230             case SDLK_s:
231                 addm_m_neighbours(MU, MV, FORWARDS);
232
233
234
235
236
237
238
239
240
241
242

```

```

243         help("Key signees selected.");
244         oprm_all(MU, MV, select_mode);
245         delm_all(MV);
246         break;
247
248     case SDLK_u: help("TODO: redo."); break;
249
250     case SDLK_w: help("TODO: save bitmap image."); break;
251
252     case SDLK_x: help("TODO: set key redraw rate."); break;
253
254     case SDLK_z:
255         if (!selection_empty())
256         {
257             center_frame_m(MU);
258             help("Zoom centered on selection");
259         } else {
260             center_frame_m(MNONE);
261             help("Zoom centered on all keys.");
262         }
263         break;
264
265     case SDLK_LSHIFT:
266     case SDLK_RSHIFT:
267     case SDLK_LCTRL:
268     case SDLK_RCTRL:
269     case SDLK_LALT:
270     case SDLK_RALT:
271     case SDLK_ESCAPE:
272         // ignore
273         break;
274
275     default:
276         help("No command bound to this key.");
277         break;
278 }
279 } else {
280     switch (event.key.keysym.sym)
281     {
282     case SDLK_a:
283         addm_all(MU); help("All keys selected.");
284         break;
285
286     case SDLK_c:
287         user_input("Please enter the color to use for selected keys (hexa):",
288                     &opsel_by_color, "0x", 2 );
289         break;
290
291     case SDLK_d:
292         // compute distances
293         nfrom = count_m(MF);
294         nto = count_m(MT);
295
296         if (nfrom && nto)
297         {
298             remove_tooltip();
299             total_dist = total_dists(100, nfrom, 0, 0, 0.5);
300             total_gdist = total_gdists(100, nfrom, 0, 0.5, 1);
301             if (total_dist > 0 || total_gdist > 0)
302             {
303                 sprintf(msg, "Total dist : %lf, average dist %lf — Total gdist : %lf, average
304                             gdist %lf — gdist/dist=%lf", total_dist, total_dist/(nfrom*nto),
305                             total_gdist, total_gdist/(nfrom*nto), total_gdist/total_dist);
306             } else {
307                 sprintf(msg, "Abort.");
308             }
309         } else {
310             sprintf(msg, "Select from and to keys with 'f' first.");
311         }
312         help(msg);
313         break;
314
315     case SDLK_e:
316         user_input("Please enter a TLD:", &opsel_by_tld, ">", 0 );
317         break;
318
319     case SDLK_f:
320         addm_selected(MF); help("Keys tagged as From.");
321         break;
322
323     case SDLK_g:
324         addm_selected(MA); help("Key accel displayed for selected keys.");
325         break;

```

```
324     case SDLK_h: help("TODO: help."); break;
325
326     case SDLK_i:
327         addm_selected(MI); help("Key IDs displayed for selected keys.");
328         break;
329
330     case SDLK_k:
331         user_input("Please enter a key ID:", &opsel_by_id, "0x", 2);
332         break;
333
334     case SDLK_l:
335         do_all_selected(&update_key_accel);
336         break;
337
338     case SDLK_m:
339         do_all_m(&move_key_step, MU);
340         break;
341
342     case SDLK_n:
343         addm_selected(MN); help("Keys names displayed for selected keys.");
344         break;
345
346     case SDLK_q:
347         // exit
348         return -1;
349         break;
350
351     case SDLK_r:
352         user_input("Please enter the number of random keys to select:", &opsel_rand, "",
353             0);
354         break;
355
356     case SDLK_s:
357         addm_m_neighbours(MU, MV, BACKWARDS);
358         help("Key signers selected.");
359         oprm_all(MU, MV, select_mode);
360         delm_all(MV);
361         break;
362
363     case SDLK_t:
364         sprintf(msg, "%lu keys out of %lu selected.", count_m(MU), num_keys);
365         help(msg);
366         break;
367
368     case SDLK_u: help("TODO: undo."); break;
369
370     case SDLK_v:
371         notm_all(MU); help("Selection inverted.");
372         break;
373
374     case SDLK_w: help("TODO: save"); break;
375
376     case SDLK_x:
377         if (auto_recalc)
378             {
379                 auto_recalc=0; help("Auto-recalc disabled.");
380             }
381         else {
382             auto_recalc=1; help("Auto-recalc enabled.");
383         }
384         break;
385
386     case SDLK_z:
387         if (!selection_empty())
388             {
389                 reset_frame_m(MU); help("Zoom fitted to selection.");
390             }
391         else {
392             reset_frame_m(MNONE); help("Zoom fitted to all keys.");
393         }
394         break;
395
396     case SDLK_SLASH:
397         select_mode = SELECT_SET;
398         help("Next selection commands will define selection.");
399         break;
400
401     case SDLK_PLUS:
402         select_mode = SELECT_UNION;
403         help("Next selection commands will be added to selection.");
404         break;
405
406     case SDLK_MINUS:
407         select_mode = SELECT_REMOVE;
408         help("Next selection commands will be removed from selection.");
```

```

407         break;
408
409     case SDLK_ASTERISK:
410         select_mode = SELECT_INTERSECTION;
411         help("Next selection commands will be intersected with selection.");
412         break;
413
414     case SDLK_BACKSLASH:
415         select_mode = SELECT_XOR;
416         help("Next selection commands will be XORed with selection.");
417         break;
418
419
420     case SDLK_WORLD_95:
421         // TODO: for some weird reason, linking fails unless this command appears
422                 somewhere...
423         SDL_Input.InputString(NULL, NULL, 0,0);
424         break;
425
426     case SDLK_LSHIFT:
427     case SDLK_RSHIFT:
428     case SDLK_LCTRL:
429     case SDLK_RCTRL:
430     case SDLK_LALT:
431     case SDLK_RALT:
432     case SDLK_ESCAPE:
433         // ignore
434         break;
435
436     default:
437         help("No command bound to this key.");
438         break;
439     }
440 }
441 break;
442
443
444 case SDLMOUSEBUTTONDOWN:
445     switch (event.button.button)
446     {
447     case SDL_BUTTON_RIGHT:
448         // move keys
449         if (!selection_empty())
450         {
451             // start moving current selection
452             moving=1;
453         } else {
454             // we have no selection to move around
455             if ((key = find_closest(disp2real_x(event.button.x), disp2real_y(event.button.y),
456                                     MIN_DIST*(disp2real_x(1) - disp2real_x(0))*(disp2real_y(1) - disp2real_y(0)))) !=
457                 num.keys)
458             {
459                 // mouse pointer is close to a key, so we select it and start moving it
460                 opsel_by_id_p(vertices[key].id);
461                 moving = 2; // to remember that the key should be unselected when right button is
462                 released (it wasn't selected beforehand, because selection_empty() returned
463                 TRUE)
464             } else {
465                 // no selection and no nearby key, so we can't start moving anything
466             }
467         }
468
469         if (moving)
470         {
471             mouse_x = event.button.x;
472             mouse_y = event.button.y;
473         }
474
475         break;
476
477     case SDL_BUTTON_LEFT:
478         // record current pointer coordinates
479         mouse_x = event.button.x;
480         mouse_y = event.button.y;
481         // we don't know yet if user wants to draw a rectangle or select just one key
482         mouse.byrect=RECT_MAYBE;
483         break;
484
485     case SDL_BUTTON_MIDDLE:
486         mouse_mx = event.button.x;
487         mouse_my = event.button.y;

```



```

486     mouse_mx_r = ax;
487     mouse_my_r = ay;
488
489     // start moving view
490     mouse_middle = 1;
491     break;
492
493     case SDL_BUTTON_WHEELDOWN:
494         // zoom out
495         zoom(event.button.x, event.button.y, -ZFACTOR);
496         break;
497
498     case SDL_BUTTON_WHEELUP:
499         // zoom in
500         zoom(event.button.x, event.button.y, ZFACTOR);
501         break;
502
503     default:
504         help("No command bound to this button.");
505         break;
506 }
507 break;
508
509 case SDL_MOUSEBUTTONDOWN:
510     switch (event.button.button)
511     {
512         case SDL_BUTTON_RIGHT:
513             if (moving == 2) delm_all(MU); // unselect the key which was temporarily selected
514             moving = 0; // stop moving keys
515             break;
516
517         case SDL_BUTTON_LEFT:
518
519             dx = event.button.x - mouse_x;
520             dy = event.button.y - mouse_y;
521
522             if (mouse_byrect != RECT_YES)
523             {
524                 // user selected one key
525                 // TODO: avoid already selected keys for operator union, etc.
526                 // find closest key and select it
527                 opsel_by_id_p(vertices[find_closest(disp2real_x(event.button.x),
528                                     disp2real_y(event.button.y), MIN_DIST*(disp2real_x(1) -
529                                     disp2real_x(0))*(disp2real_y(1) - disp2real_y(0)))]).id);
530             } else {
531                 // select all keys in region between pointer coordinates at MOUSEBUTTONDOWN and
532                 // pointer coordinates at MOUSEBUTTONUP
533                 opsel_by_region_p(disp2real_x(event.button.x), disp2real_y(event.button.y),
534                                 disp2real_x(mouse_x), disp2real_y(mouse_y));
535             }
536
537             mouse_byrect = RECT_NO;
538             break;
539
540         case SDL_BUTTON_MIDDLE:
541             // stop moving view
542             mouse_middle = 0;
543             break;
544     }
545     break;
546
547 case SDL_MOUSEMOTION:
548     pointer_x = event.motion.x;
549     pointer_y = event.motion.y;
550
551     if (mouse_middle)
552     {
553         // change display position
554         ddx = bx - ax;
555         ddy = by - ay;
556
557         ax -= -mouse_mx_r + disp2real_x(-mouse_mx + event.motion.x);
558         ay -= -mouse_my_r + disp2real_y(-mouse_my + event.motion.y);
559
560         bx = ax + ddx;
561         by = ay + ddy;
562     }
563
564     if (moving)
565     {
566         // move keys around
567         delta_x = - ((disp2real_x(mouse_x)) - disp2real_x(event.motion.x));
568         delta_y = - ((disp2real_y(mouse_y)) - disp2real_y(event.motion.y));
569     }

```

```
566     do_all_selected(&moving_adjust);
567
568     mouse_x = event.motion.x;
569     mouse_y = event.motion.y;
570 }
571
572 if (mouse_byrect == RECTMAYBE)
573 {
574     dx = event.button.x - mouse_x;
575     dy = event.button.y - mouse_y;
576
577     if (dx*dx + dy*dy < SELECTION_DELTA)
578         mouse_byrect = RECT.YES; // user has moved pointer too far, he probably intends a
579         rectangle
580 }
581
582 if (!moving)
583 {
584     // display a tooltip
585     key = find_closest(dis2real_x(event.motion.x), dis2real_y(event.motion.y),
586         MIN_DIST*(dis2real_x(1) - dis2real_x(0))*(dis2real_y(1) - dis2real_y(0)));
587     if (!had_mn) delm(last_key, MN);
588     if (!had_mi) delm(last_key, MI);
589     vertices[last_key].color = had_color;
590     last_key = key;
591     // we need to switch MN, MI and MU temporarily to allow for caption drawing
592     had_mn = hasm(key, MN);
593     had_mi = hasm(key, MI);
594     had_color = vertices[key].color;
595     addm(key, MN);
596     addm(key, MI);
597     vertices[key].color = white;
598     draw_caption(key);
599 }
600
601 break;
602
603 case SDL_VIDEORESIZE:
604     apply_resize(event);
605     break;
606
607 case SDL_QUIT:
608     // exit
609     return -1;
610     break;
611 }
612
613 return 1;
614 }
```

Listing 9 – graphics.c : Affichage graphique

```

1  #include "main.h"
2
3
4  double ax, ay, bx, by; // frame
5
6  extern double delta_x, delta_y;
7  extern char inputting;
8
9  char tooltip=0;
10 char tooltip_time=0;
11 void *input_callback;
12
13 int width=WIDTH;
14 int height=HEIGHT;
15
16 // for rectangle selection
17 int mouse_x, mouse_y;
18 // to display rectangle on mouse move
19 char mouse_byrect=0;
20
21 // for mouse button drag and drop
22 int mouse_mx, mouse_my;
23 // matching real coordinates
24 double mouse_mx_r, mouse_my_r;
25 // to refresh on mouse move
26 char mouse_middle=0;
27
28 char status[500];
29
30 TTF_Font *font = NULL;
31 TTF_Font *small_font = NULL;
32
33 SDL_Surface *screen = NULL;
34 SDL_Input_TTF *ttf = NULL;
35 SDL_Rect inputpos;
36
37 extern int id(int a);
38 extern int is_null(void* a);
39
40 extern double get_kx(unsigned long pos);
41 extern double get_ky(unsigned long pos);
42 extern double get_kax(unsigned long pos);
43 extern double get_kay(unsigned long pos);
44 extern void circle(unsigned long pos, double radius, double cx, double cy);
45
46 extern SDL_Color had_color;
47 SDL_Rect pos_status;
48
49 extern key vertices[MAXKEYS];
50
51 extern struct sig sigs[MAXTSIGS];
52 extern unsigned int num_sigs;
53 extern unsigned long num_keys;
54
55 extern signatures edges[MAXKEYS]; // for BACKWARDS: list of signers for each signee
56 extern signatures redges[MAXKEYS]; // for FORWARDS: list of signees for each signer
57
58 extern int pointer_x, pointer_y;
59
60
61
62 int sdl_init() { return SDL_Init(SDL_INIT_VIDEO); }
63
64 // TODO segfaults when the window is resized
65 int sdl_setvideomode() { return (screen = SDL_SetVideoMode(width, height, DEPTH, SDL_HWSURFACE |
66     SDL_RESIZABLE)); }
67
68 int load_font() { return (font = TTF_OpenFont(FONT, INTERFACE_FONT_SIZE)) != NULL; }
69 int load_small_font() { return (small_font = TTF_OpenFont(FONT, KEY_FONT_SIZE)) != NULL; }
70
71 int load_input_field() { return (ttf = SDL_Input_TTF_Create( MAXNAMELEN, font, white, &black,
72     inputpos, SDL_INPUT_TTF_SOLID )) != NULL; }
73
74 enum error graphics_init()
75 {
76     // Run all initialisation functions, and fail if one of them fails.
77     try(&sdl_init, &id, ERR_SDL_INIT, "Erreur d'initialisation de la SDL.\n");
78     try(&TTF_Init, &id, ERR_TTF_INIT, "Erreur d'initialisation de la SDL.TTF.\n");
79     try(&sdl_setvideomode, &is_null, ERR_VIDEO_INIT, "Erreur d'initialisation du mode
80         graphique.\n");

```

```
79     SDL_WM_SetCaption("Key visu", NULL);
80     try(&load_font, &is_null, ERR_FONT, "Erreur de chargement de la police.\n");
81     try(&load_small_font, &is_null, ERR_FONT, "Erreur de chargement de la police.\n");
82     try(&load_input_field, &is_null, ERR_SDL_INPUT_INIT, "Erreur de chargement de la
        SDL_Input_TTF.\n");
83
84     SDL_EnableKeyRepeat( SDL_DEFAULT_REPEAT_DELAY, SDL_DEFAULT_REPEAT_INTERVAL );
85
86     SDL_EnableUNICODE( 1 );
87
88     return ERR_NONE;
89 }
90
91
92 void globals_init()
93 {
94     pos_status.x = 0;
95     pos_status.y = 0;
96
97     inputpos.x = 0;
98     inputpos.y = 20;
99     black.r = 0; black.g = 0; black.b = 0;
100    white.r = 0xff; white.g = 0xff; white.b = 0xff;
101    red.r = 0xff; red.g = 0; red.b = 0;
102    green.r = 0; green.g = 0xff; green.b = 0;
103    blue.r = 0; blue.g = 0; blue.b = 0xff;
104    had_color.r = 0; had_color.g = 0; had_color.b = 0xff;
105 }
106
107
108 char valid_coord(int x, int y)
109 {
110     // Check if coordinates are valid.
111     return (x>=0) && (x < width) && (y>0) && (y < height);
112 }
113
114
115 void pixel_draw(int x, int y, Uint8 r, Uint8 g, Uint8 b, char size)
116 {
117     // Draw a key.
118
119     Uint32 *p;
120     int i, j;
121
122     for (i=-(size/2); i<(size/2+1); i++)
123         for (j=-(size/2); j<(size/2+1); j++)
124             if (valid_coord(x+i, y+j))
125             {
126                 p = screen->pixels + (y+j) * screen->pitch + (x+i) * screen->format->BytesPerPixel;
127                 *p|=SDL_MapRGB(screen->format, r, g, b);
128             }
129 }
130
131
132 void set_status(char * stat)
133 {
134     if (!tooltip)
135     {
136         if (strcmp(status, stat))
137             printf("%s\n", stat);
138         strcpy(status, stat);
139     }
140 }
141
142
143 void update_status()
144 {
145     writeTxt(screen, status, 0, 0, white, font);
146 }
147
148
149 void remove_tooltip()
150 {
151     tooltip=TOOLTIP_NONE;
152     tooltip_time=0;
153 }
154
155
156 void set_tooltip(char t, char * ttip)
157 {
158     remove_tooltip();
159     set_status(ttip);
160     tooltip = t;
161     tooltip_time = TOOLTIP_TIME;
```

```
162 }
163
164
165 void update_tooltip()
166 {
167     if (tooltip)
168     {
169         tooltip_time--;
170         if (!tooltip_time)
171             tooltip=TOOLTIP.NONE;
172     }
173 }
174
175
176 void update_input()
177 {
178     if (inputting)
179         SDL_Input_TTF_Display( ttf, screen, NULL );
180 }
181
182
183 void fit_frame(unsigned long i)
184 {
185     // Adjust frame to fit a key.
186
187     if (get_kx(i) < ax || ax < 0)
188         ax = get_kx(i) - KEYSIZE;
189     if (get_ky(i) < ay || ay < 0)
190         ay = get_ky(i) - KEYSIZE;
191     if (get_kx(i) > bx)
192         bx = get_kx(i) + KEYSIZE;
193     if (get_ky(i) > by)
194         by = get_ky(i) + KEYSIZE;
195 }
196
197
198 void reset_frame_m(int val) {
199     // Adjust frame to fit keys with mark val.
200
201     double dx, dy;
202     double bax=ax, bbx=bx, bay=ay, bby=by;
203
204     bx=by=0;
205     ax=ay=-1;
206     do_all_m(&fit_frame, val);
207
208     dx = bx-ax;
209     dy = by-ay;
210     ax-=dx*FITOFFSET;
211     bx+=dx*FITOFFSET;
212     ay-=dy*FITOFFSET;
213     by+=dy*FITOFFSET;
214
215     // in case it didn't work (no key with mark, etc.)
216     if (bx-ax <= 0 || by-ay <= 0)
217     {
218         ax = bax;
219         ay = bay;
220         bx = bbx;
221         by = bby;
222     }
223 }
224
225 void reset_frame() { reset_frame_m(MNONE); }
226
227
228 void center_frame_m(int val)
229 {
230     // Center frame around keys with mark val (without changing scale).
231
232     double bax=ax, bbx=bx, bay=ay, bby=by;
233     double dx, dy;
234
235     // first, we adujst frame
236     reset_frame_m(val);
237
238     // then, we restore previous scale around the new center
239     dx = (bx + ax)/2 - (bbx + bax)/2;
240     dy = (by + ay)/2 - (bby + bay)/2;
241
242     ax = bax + dx;
243     ay = bay + dy;
244     bx = bbx + dx;
245     by = bby + dy;
```

```
246 }
247
248 void center_frame() { center_frame_m(MNONE); }
249
250
251 void draw_rect(int x1, int y1, int x2, int y2, char r, char g, char b)
252 {
253     // Draw a rectangle.
254
255     int i;
256     int xa, xb, ya, yb;
257
258     SDL_LockSurface(screen);
259
260     if (x1<x2) {xa = x1; xb = x2;} else {xa = x2; xb=x1;}
261     if (y1<y2) {ya = y1; yb = y2;} else {ya = y2; yb=y1;}
262
263     for (i = xa; i<=xb; i++)
264     {
265         pixel_draw(i, ya, r, g, b, SELECTION_RECT.SIZE);
266         pixel_draw(i, yb, r, g, b, SELECTION_RECT.SIZE);
267     }
268
269     for (i = ya; i<=yb; i++)
270     {
271         pixel_draw(xa, i, r, g, b, SELECTION_RECT.SIZE);
272         pixel_draw(xb, i, r, g, b, SELECTION_RECT.SIZE);
273     }
274
275     SDL_UnlockSurface(screen);
276 }
277
278
279 void redraw()
280 {
281     // Redraw everything except the graph.
282
283     int x, y;
284
285     update_tooltip();
286     update_status();
287     update_input();
288
289     if(mouse_byrect == 1)
290     {
291         // draw rectangle
292         SDL_GetMouseState(&x, &y);
293         draw_rect(x, y, mouse_x, mouse_y, 0, 80, 189);
294     }
295
296     SDL_Flip(screen);
297 }
298
299
300 void accel_draw(unsigned long i)
301 {
302     // Draw acceleration for a key.
303     // TODO: avoid magic numbers, find a more efficient way, antialiasing etc.
304
305     unsigned long j;
306     Uint32 *p;
307     float adx, ady;
308     int x, y;
309
310     SDL_LockSurface(screen);
311
312     for (j=0; j<10000; j++)
313     {
314         adx = ((get_kx(i)+j*get_kax(i)/1000-ax)/(bx-ax));
315         ady = ((get_ky(i)+j*get_kay(i)/1000-ay)/(by-ay));
316
317         x = (width-1)*adx;
318         y = (height-1)*ady;
319
320         if (valid_coord(x, y))
321         {
322             p = screen->pixels + y * screen->pitch + x * screen->format->BytesPerPixel;
323
324             *p|=SDL_MapRGB(screen->format, 255, 0,0);
325         }
326     }
327
328     SDL_UnlockSurface(screen);
329 }
```

```
330
331
332 void draw_caption(unsigned long i)
333 {
334     // Draw caption for a key.
335     // TODO 2: avoid overlapping, etc.
336
337     char label[500];
338     float adx, ady;
339     int x, y;
340
341     adx = ((get_kx(i)-ax)/(bx-ax));
342     ady = ((get_ky(i)-ay)/(by-ay));
343     x = (width-1)*adx;
344     y = (height-1)*ady;
345
346     if (valid_coord(x, y))
347     {
348         if (hasm(i,MI) && hasm(i,MN))
349             sprintf((char *) &label, "%x - %s", vertices[i].id, vertices[i].name);
350         else if (hasm(i, MI))
351             sprintf((char *) &label, "%x", vertices[i].id);
352         else
353             sprintf((char *) &label, "%s", vertices[i].name);
354         if (hasm(i, MU)) writeTxt(screen, label, x, y, white, small_font);
355         else writeTxt(screen, label, x, y, vertices[i].color, small_font);
356     }
357 }
358
359
360 void apply_resize(SDL_Event event)
361 {
362     width = event.resize.w;
363     height = event.resize.h;
364     printf("%d %d\n", width, height);
365     reset_frame();
366 }
367
368
369 void graph_redraw(double ep)
370 {
371     // Redraw graph.
372
373     unsigned long i;
374     float adx, ady;
375     int x, y;
376
377     char label[500];
378
379     SDL_LockSurface(screen);
380     SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 0, 0, 0));
381
382     for(i=0; i<num_keys; i++)
383     {
384         adx = ((get_kx(i)-ax)/(bx-ax));
385         ady = ((get_ky(i)-ay)/(by-ay));
386         x = (width-1)*adx;
387         y = (height-1)*ady;
388
389         if (valid_coord(x, y))
390         {
391             pixel_draw(x, y, vertices[i].color.r, vertices[i].color.g, vertices[i].color.b, DOT_SIZE);
392
393             if (hasm(i, MU))
394                 pixel_draw(x, y, 0xff, 0xff, 0xff, DOT_SIZE);
395         }
396     }
397
398     SDL_UnlockSurface(screen);
399
400     do_all_m(&accel_draw, MA);
401     do_all_m(&draw_caption, MI);
402     do_all_m(&draw_caption, MN);
403
404     if (ep)
405     {
406         sprintf((char*) &label, "EP: %le", ep);
407         set_status(label);
408         //printf("%s\n", eps);
409     }
410 }
411
412
413 void redraw_all(double ep)
```

```

414 {
415     graph_redraw(ep);
416     redraw();
417 }
418
419
420 void in_circle(char* radius)
421 {
422     unsigned long i;
423     for (i=0; i<num_keys; i++)
424         if (hasm(i, MU))
425             circle(i, disp2real_x(atoi(radius)) - disp2real_x(0), disp2real_x(pointer_x),
426                     disp2real_y(pointer_y));
427 }
428
429 void opsel_by_color(char* color)
430 {
431     unsigned long color_p;
432     SDL_Color s_color;
433
434     color_p = strtol(color, NULL, 16);
435     s_color.r = color_p >> 16;
436     s_color.g = ((color_p ^ (s_color.r << 16)) >> 8);
437     s_color.b = (color_p ^ (s_color.r << 16)) ^ (s_color.g << 8);
438     color_m(s_color, MU);
439     printf("Marked with color %x\n", color_p);
440     delm_all(MV);
441 }
442
443 void color_m(SDL_Color color, char mark)
444 {
445     // Color by mark.
446
447     unsigned long i;
448     for (i=0; i<num_keys; i++)
449         if (hasm(i, mark))
450             vertices[i].color = color;
451 }
452
453
454 void stop_input()
455 {
456     inputting=0;
457     if (tooltip == TOOLTIP_INPUT)
458         remove_tooltip();
459 }
460
461
462 double disp2real_x(double disp_x)
463 {
464     // converts an on-screen coordinate to a real coordinate
465     return ax+(disp_x/width)*(bx-ax);
466 }
467
468 double disp2real_y(double disp_y)
469 {
470     return ay+(disp_y/height)*(by-ay);
471 }
472
473
474 void zoom(int x, int y, double factor)
475 {
476     ax = ax + (((double) bx - ax) * (((double) x) / ((double) width)) / factor);
477     ay = ay + (((double) by - ay) * (((double) y) / ((double) height)) / factor);
478     bx = bx + (((double) bx - ax) * (1. - (((double) x) / ((double) width))) / factor);
479     by = by - (((double) by - ay) * (1. - (((double) y) / ((double) height))) / factor);
480 }
481
482
483 void help(char* msg) { set_tooltip(TOOLTIP_HELP, msg); }
484 void label(char* msg) { set_tooltip(TOOLTIP_INPUT, msg); }
485
486
487 void user_input(char* caption, void (*fun)(char*), char* initial, int initial_cursor)
488 {
489     // Start user input.
490     label(caption);
491     input_callback = fun;
492     inputting = 1;
493     SDL_Input_SetText(ttf->input, initial);
494     SDL_Input_SetCursorIndex(ttf->input, initial_cursor);
495 }
496

```



```
497
498 int writeTxt(SDL_Surface* screen, char *message, signed int x, signed int y, SDL_Color color,
499           TTF_Font* font)
500 {
501     // Write some text.
502     int rslt=0;
503
504     if(message != NULL)
505     {
506         SDL_Surface* txt = NULL;
507         SDL_Rect position;
508
509         txt = TTF_RenderText_Solid(font, message, color);
510         position.x = x;
511         position.y = y;
512         SDL_BlitSurface(txt, NULL, screen, &position);
513
514         SDL_FreeSurface(txt);
515     }
516     else rslt = 1;
517
518     return rslt;
519 }
520
521 int progress_bar(double x)
522 {
523     // Update progress bar.
524
525     SDL_Event event;
526     SDL_Surface *bar;
527
528     printf("Progress is %lf.\n", x);
529     if (x>=0 && x<=1)
530     {
531         bar = SDL_CreateRGBSurface(SDL_HWSURFACE, (int) (x * (width-1))+1, 20, 32, 0, 0, 0, 0);
532         SDL_FillRect(bar, NULL, SDL_MapRGB(screen->format, 200, 200, 200));
533         SDL_BlitSurface(bar, NULL, screen, &pos_status);
534         SDL_FreeSurface(bar);
535         SDL_Flip(screen);
536     }
537     else printf("ERROR: progress_bar value is not within [0, 1] range (got %lf)\n", x);
538
539     // allow user to cancel
540     while (SDL_PollEvent(&event))
541     {
542         switch(event.type)
543         {
544             case SDL_KEYDOWN:
545                 if (event.key.keysym.sym == SDLK_ESCAPE)
546                     return (-1);
547         }
548     }
549     return 0;
550 }
551
552 void graphics_end()
553 {
554     TTF_CloseFont(font);
555     TTF_CloseFont(small_font);
556     TTF_Quit();
557     SDL_Quit();
558 }
559
```

Listing 10 – misc.c : Fonctions diverses

```

1  #include "main.h"
2
3
4  extern unsigned long num_keys;
5  extern key vertices[MAXKEYS];
6  extern signatures edges[MAXKEYS];
7  extern signatures redges[MAXKEYS];
8
9  extern double get_kx(unsigned long pos);
10 extern double get_ky(unsigned long pos);
11
12
13
14 void try(void* (*fun)(void), int (*predicate)(void*), enum error errcode, char* errdesc)
15 {
16     // Try doing fun, and check its return value with predicate. If predicate returns TRUE, fun has
17     // failed: exit with errcode and message errdesc.
18     if ((*predicate)((*fun)()))
19     {
20         fprintf(stderr, errdesc);
21         exit(errcode);
22     }
23 }
24
25 int is_null(void* a) { return a==NULL; }
26
27 void do_all(void (*fun)(unsigned long int))
28 {
29     // Apply fun to every key.
30     unsigned long i;
31
32     for (i=0; i<num_keys; i++)
33         (*fun)(i);
34 }
35
36 // different versions according to fun's parameters
37 // TODO: isn't there a better way?
38 void do_all_c(void (*fun)(unsigned long int, char), char c)
39 {
40     unsigned long i;
41     for (i=0; i<num_keys; i++)
42         (*fun)(i, c);
43 }
44
45 void do_all_d(void (*fun)(unsigned long int, double), double c)
46 {
47     unsigned long i;
48     for (i=0; i<num_keys; i++)
49         (*fun)(i, c);
50 }
51
52 void do_all_m(void (*fun)(unsigned long int), char m)
53 {
54     // Apply to all keys with mark m.
55     unsigned long i;
56
57     for (i=0; i<num_keys; i++)
58         if (hasm(i, m))
59             (*fun)(i);
60 }
61
62 void do_all_selected(void (*fun)(unsigned long int)) { do_all_m(fun, MU); }
63
64 unsigned long get_pos_from_id(unsigned long id)
65 {
66     // Get key index in vertices from its OpenPGP id.
67     unsigned long i;
68
69     for (i=0; i<num_keys; i++)
70         if (vertices[i].id == id)
71             return i;
72
73     printf("ERROR: invalid key requested.\n");
74     return MAXKEYS;
75 }
76
77
78
79
80

```

```
81 unsigned long read_ul(FILE *f)
82 {
83     // Read an unsigned long from a file.
84
85     unsigned long a=0;
86     a |= (fgetc(f) << 24); if (feof(f)) return 0;
87     a |= (fgetc(f) << 16); if (feof(f)) return 0;
88     a |= (fgetc(f) << 8); if (feof(f)) return 0;
89     a |= (fgetc(f));
90     return a;
91 }
92
93 void trim_trailing_newline(unsigned long pos)
94 {
95     int n = strlen(vertices[pos].name);
96     if (vertices[pos].name[n - 1] == '\n')
97         vertices[pos].name[n - 1] = '\0';
98 }
99
100 unsigned long get_type_from_sig_data(unsigned long sig_data)
101 {
102     // Handles the signature data from the Wotsap file.
103     return sig_data >> 28;
104 }
105 unsigned long get_id_from_sig_data(unsigned long sig_data)
106 {
107     return (get_type_from_sig_data(sig_data) << 28) ^ sig_data;
108 }
109
110 void name_key_p(unsigned long pos)
111 {
112     // Display key name, and show if it is marked or not.
113
114     if (is_mark(pos)) printf("(*) "); else printf(" ");
115     printf("%x - %s (%lu)\n", vertices[pos].id, vertices[pos].name, pos);
116 }
117
118 void report_on_key(unsigned long id)
119 {
120     // Display some info about a key.
121
122     unsigned long pos = get_pos_from_id(id);
123     unsigned long n=edges[pos].num;
124     struct signature * s = edges[pos].head;
125
126     printf("Key %lu is %lu in WOT database.\n", id, pos);
127     printf("It belongs to %s.\n", vertices[pos].name);
128     printf("It has been signed by %lu keys:\n", n);
129
130     while (s)
131     {
132         name_key_p(s->s->f); s = s->next;
133     }
134
135     printf("It has signed %lu keys:\n", n);
136
137     s = redges[pos].head;
138     while (s)
139     {
140         name_key_p(s->s->t); s = s->next;
141     }
142
143     breadth_explore(pos, FORWARDS, 1); breadth_explore(pos, BACKWARDS, 1);
144 }
145
146
147 char get_tld(unsigned long pos, char* tld, char mlen)
148 {
149     // Get TLD of a key.
150
151     int dot_pos= strlen(vertices[pos].name);
152     int i, tld_len;
153     if (vertices[pos].name[dot_pos-1] != '>')
154         return 1; // no email address
155     for(i=0; i<mlen; i++)
156         if(vertices[pos].name[dot_pos-i] == '.')
157             break;
158     tld_len = i;
159     dot_pos = dot_pos - tld_len;
160     for (i=0; i<tld_len; i++)
161         tld[i] = vertices[pos].name[dot_pos+i];
162     tld[tld_len] = '\0';
163     return 0;
164 }
```

```
165 }
166
167
168 unsigned long find_closest_m(double x, double y, int val, double dist)
169 {
170     // Find closest key to a point with a specific mark (compute all distances and check which one
171     // is smaller)
172     // dist is minimum distance for matching
173
174     double dx, dy, d;
175     unsigned long best=num_keys; // if nothing is found
176     unsigned long i;
177
178     for (i=0; i<num_keys; i++)
179     {
180         if (hasm(i, val))
181         {
182             dx = get_kx(i) - x;
183             dy = get_ky(i) - y;
184             d = dx*dx + dy*dy;
185             if (d < dist)
186             {
187                 best = i; dist = d;
188             }
189         }
190     }
191
192     return best;
193 }
194
195 unsigned long find_closest(double x, double y, double dist) { return find_closest_m(x, y, MNONE,
196     dist); }
197
198 void strip_gt(char* tld)
199 {
200     // Strip the '>' from "<foo@example.com>".
201
202     int i=0;
203
204     while (tld[i] != 0)
205     {
206         i++;
207         if (i>0 && tld[i-1] == '>')
208             tld[i-1] = 0;
209     }
210 }
211
212 void circle(unsigned long pos, double radius, double cx, double cy)
213 {
214     // Put key pos at a random position in a circle.
215
216     double rnd;
217
218     rnd = rand();
219
220     vertices[pos].p[X] = cos(rnd) * radius + cx;
221     vertices[pos].p[Y] = sin(rnd) * radius + cy;
222 }
```