COLLEGE OF SOCIAL & BEHAVIORAL SCIENCES
School of Information

Trust and Privacy IV: Trusting software

ISTA 488: Privacy Technologies in Context

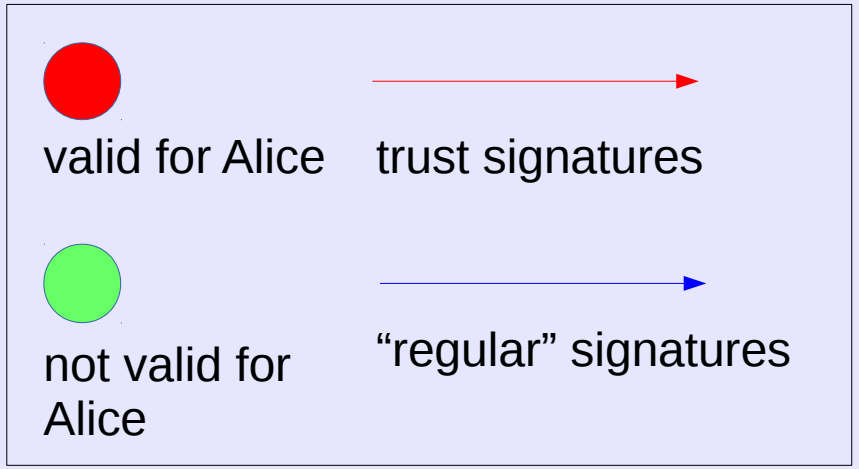
David Sidi (dsidi@email.arizona.edu)

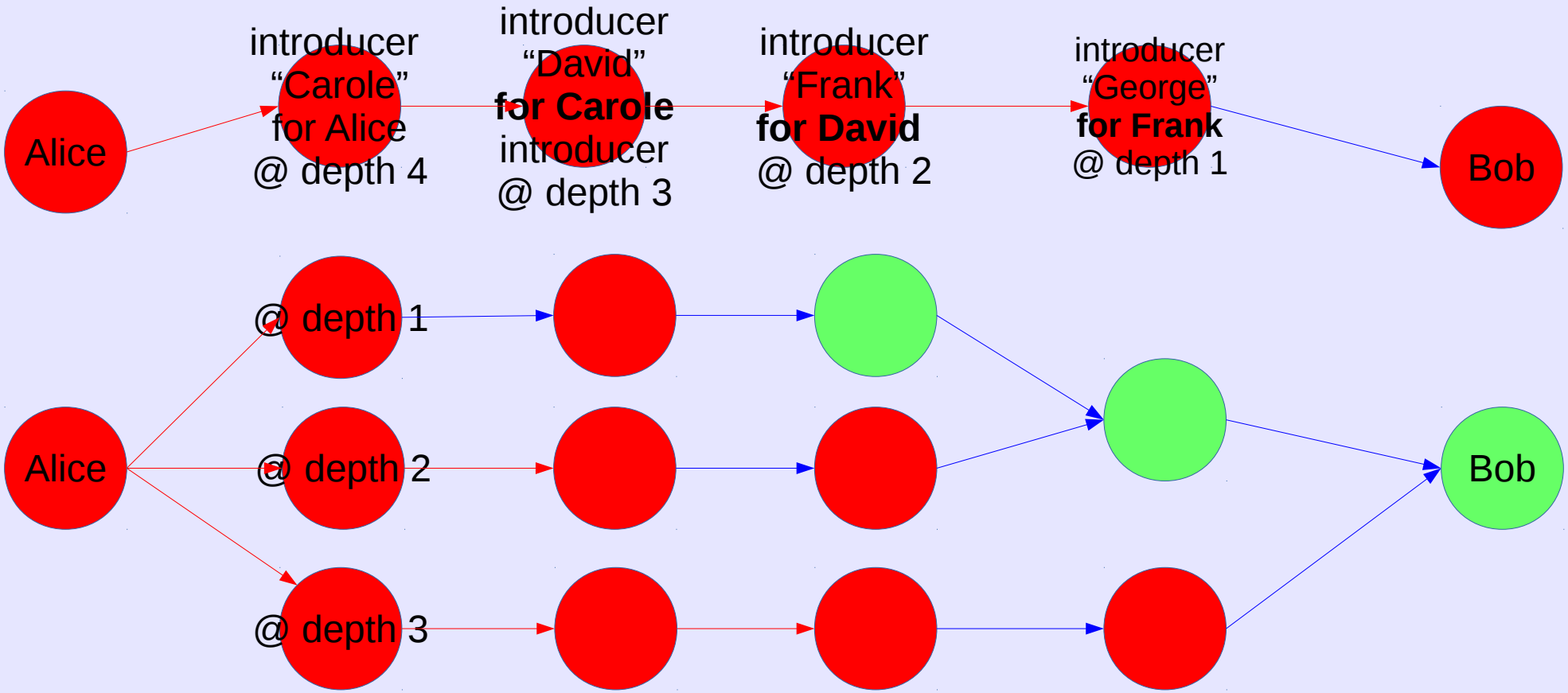
COLLEGE OF SOCIAL & BEHAVIORAL SCIENCES
School of Information

Small mention of interesting things

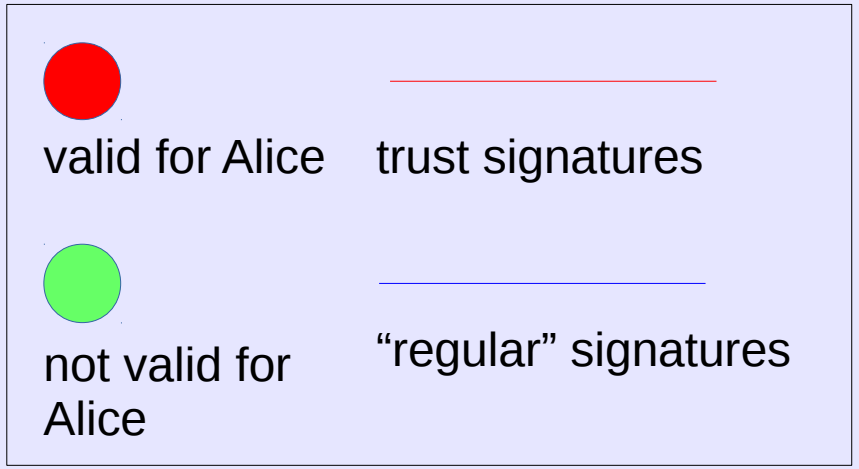
- Since it’s tricky, and I’ve received some questions: let’s review the WoT again, looking at the most indirect case of a valid key that is possible for gpg using the default settings

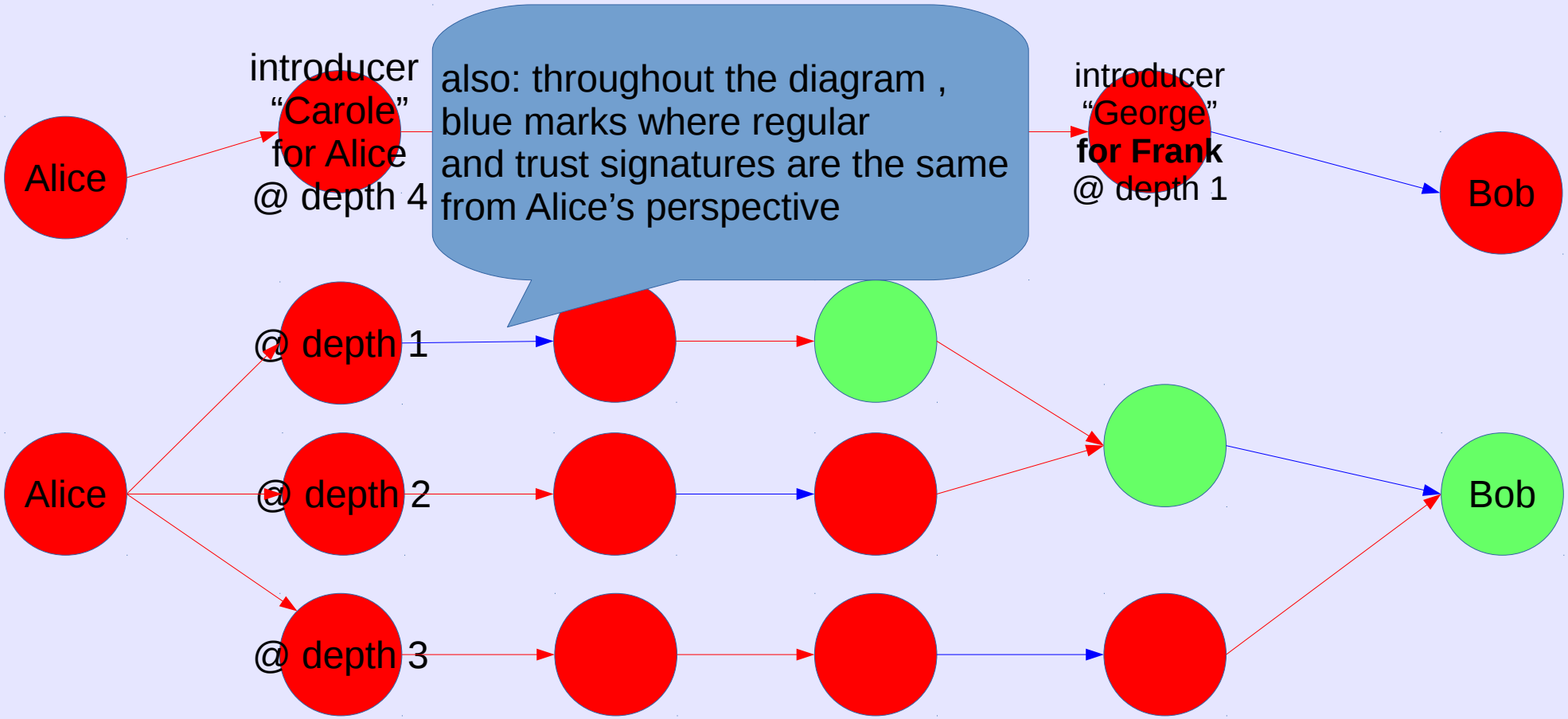
In words: valid keys are either signed (or "certified") by you, or signed by someone who is at most k-1 steps from an introducer whom you "trust at depth k." The top diagram is the *most indirect case possible with the default settings of gpg: the total path length between Alice and Bob is 5.*





In words: valid keys are either signed (or "certified") by you, or signed by someone who is at most k-1 steps from an introducer whom you "trust at depth k." The top diagram is the *most indirect case possible with the default settings of gpg: the total path length between Alice and Bob is 5.*





This is highly unclear in “An advanced introduction to Gnupg”

4.7.3 Trusted Introducers

When signing a key, it is possible to indicate that the key holder should be a trusted introducer. For instance, an organization may have a single key, say `pgp@company.com`, that they use to sign all of their employees' keys. If employees sign `pgp@company.com` using a trust signature, then anyone who trusts, say, `alice@company.com`, will, as usual, consider `pgp@company.com` to be not only verified, but, due to the trust signature, a trusted introducer. Consequently, that person will also consider any keys that `pgp@company.com` signed to be verified, which, in this case, is everyone in the company. The following example illustrates this idea:

```
juliet@                alice@                  pgp@                   bob
example -- tsign --> company -- tsign --> company -- sign --> @company
.org                   .com                    .com                   .com
```

In GnuPG, Juliet doesn't actually have to use a trust signature to sign `alice@company.com`'s key: she can just use a normal signature and then set the `ownertrust` for `alice@company.com` appropriately.

This is highly unclear in “An advanced introduction to Gnupg”

at depth 2

4.7.3 Trusted

When signing a ke[illegible] possible to indicate that the key holder should be a trusted introd[illegible] For instance, an organization may have a single key, say pgp@compa[illegible]y.com, that they use to sign all of their employees' keys. If employees sign pgp@company.com using a trust signature, then anyone who trusts, say, alice@company.com, will, as usual, consider pgp@company.com to be not only verified, but, due to the trust signature, a trusted introducer. Consequently, that person will also consider any keys that pgp@company.com signed to be verified, which, in this case, is everyone in the company. The following example illustrates this idea:

```
juliet@                alice@                 pgp@                   bob
example -- tsign --> company -- tsign --> company -- sign --> @company
.org                   .com                   .com                   .com
```

In GnuPG, Juliet doesn't actually have to use a trust signature to sign alice@company.com's key: she can just use a normal signature and then set the ownertrust for alice@company.com appropriately.

Continuing from last time: the trusting trust attack

the simplest quine

...write one yourself!

the simplest quine

careful! The empty program is copyrighted

/bin/true

```
#
#     Copyright (c) 1984 AT&T
#       All Rights Reserved
#
#     THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF AT&T
#     The copyright notice above does not evidence any
#     actual or intended publication of such source code.
#
#ident        "@(#)cmd/true.sh        50.1"
```

A simple Quine

```
char s[ ] = {
        '\t',
        '0',
        '\n',
        '}',
        ';',
        '\n',
        '\n',
        '/',
        '*',
        '\n',
        (213 lines deleted)
        0
};

/*
 * The string s is a
 * representation of the body
 * of this program from '0'
 * to the end.
 */

main( )
{
        int i;

        printf("char\ts[ ] = {\n");
        for(i=0; s[i]; i++)
                printf("\t%d, \n", s[i]);
        printf("%s", s);
}
```

Here are some simple transliterations to allow a non-C programmer to read this code.

=	assignment
==	equal to .EQ.
!=	not equal to .NE.
++	increment
'x'	single character constant
"xxx"	multiple character string
%d	format to convert to decimal
%s	format to convert to string
\t	tab character
\n	newline character

FIGURE 1.

A simple quine

(not idiomatic)

```
char s[ ] = {
        '\t',
        '0',
        '\n',
        '}',
        ';',
        '\n',
        '\n',
        '/',
        '*',
        '\n',
        (213 lines deleted)
        0
};

/*
 * The string s is a
 * representation of the body
 * of this program from '0'
 * to the end.
 */

main( )
{
        int i;

        printf("char\ts[ ] = {\n");
        for(i=0; s[i]; i++)
                printf("\t%d, \n", s[i]);
        printf("%s", s);
}
Here are some simple transliterations to allow
    a non-C programmer to read this code.
=       assignment
==      equal to .EQ.
!=      not equal to .NE.
++      increment
'x'     single character constant
"xxx"   multiple character string
%d      format to convert to decimal
%s      format to convert to string
\t      tab character
\n      newline character
```

① ②

FIGURE 1.

A quine in Python

```
from string import Template

second = Template('print(${sq}from string import Template${sq})\nprint("second =
Template(${sq}" + second.template.encode(${sq}unicode_escape${sq}).decode() + "$
{sq}")\nprint(second.substitute(sq="${sq}"))')

print('from string import Template')

print("second = Template('" + second.template.encode('unicode_escape').decode()
+ "'")

print(second.substitute(sq="'"))
```

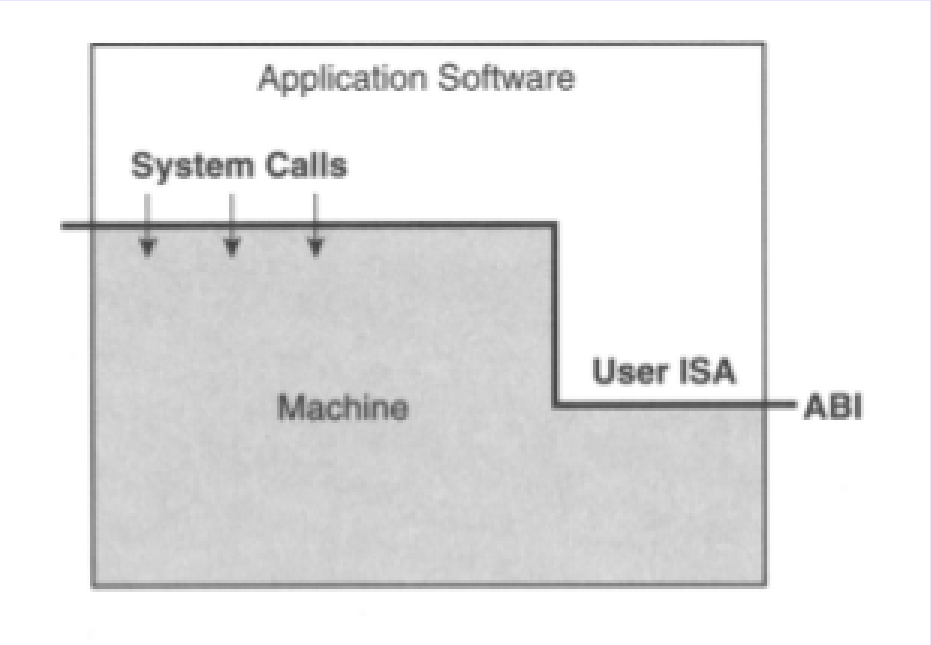
A nonexample in Python

```
#!/usr/bin/env python
import sys
import os

here = os.path.join(os.getcwd(), sys.argv[0])

with open(here) as f:
    print(''.join(line for line in f))
```

(No input allowed!)

A telling race condition

- Suppose I run one of the previous (purported) quines, but before the first line of code is run that line is changed in the program
- The quine will print out the program as modified, not the one that is running
- How do we make sure that the running code is the code that {was compiled, was run by the interpreter}?



Smith and Nair, *Virtual Machines*, p. 10

Portability and hidden behavior

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\n');
if(c == 'v')
        return('\v');
...
```

FIGURE 2.1.

```
...
c = next( );
if(c != '\\')
        return(c);
c = next( );
if(c == '\\')
        return('\\');
if(c == 'n')
        return('\ n');
if(c == 'v')
        return(11);
...
```

FIGURE 2.3.

targeting login command

```
compile(s)
char *s;
{
        ...
}
```

FIGURE 3.1.

```
compile(s)
char *s;
{
        if(match(s, "pattern")) {
                compile("bug");
                return;
        }
        ...
}
```

FIGURE 3.2.

```
compile(s)
char *s;
{
        if(match(s, "pattern1")) {
                compile ("bug1");
                return;
        }
        if(match(s, "pattern 2")) {
                compile ("bug 2");
                return;
        }
        ...
}
```

FIGURE 3.3.

emit a backdoor!

“that’s worrying... let’s audit the source code for both the login command and for the compiler we use, and recompile everything cleanly”

“that’s worrying... let’s audit the source code for both the login command and for the compiler we use, and recompile everything cleanly”

nope

```
compile(s)
char *s;
{
        ...
}
```

FIGURE 3.1.

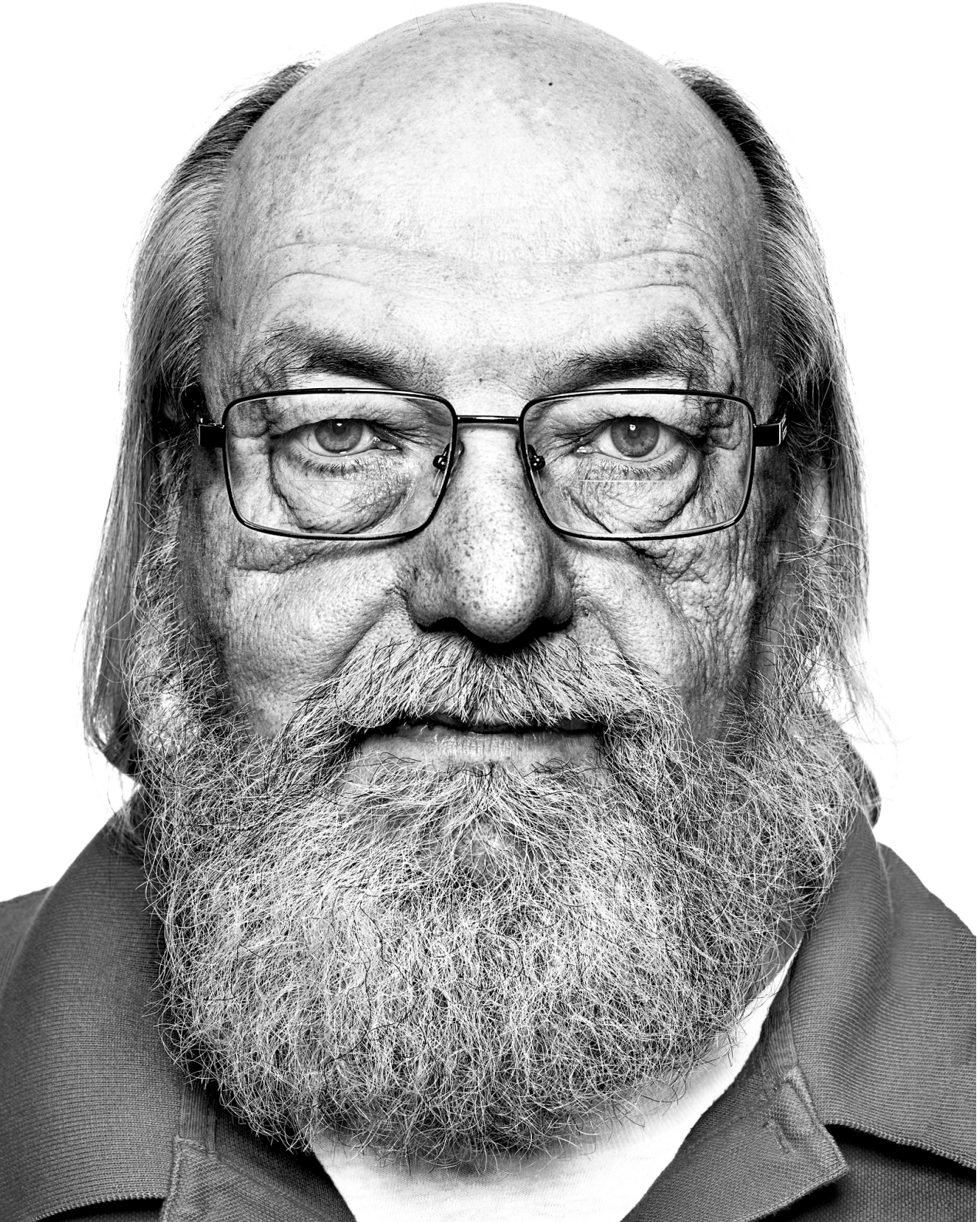
```
compile(s)
char *s;
{
        if(match(s, "pattern")) {
                compile("bug");
                return;
        }
        ...
}
```

FIGURE 3.2.

```
compile(s)
char *s;
{
        if(match(s, "pattern1")) {
                compile ("bug1");
                return;
        }
        if(match(s, "pattern 2")) {
                compile ("bug 2");
                return;
        }
        ...
}
```

FIGURE 3.3.

targeting compiler



The moral is obvious. You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me). No amount of source-level verification or scrutiny will protect you from using untrusted code.

Ken Thompson, ACM Turing Award Speech, “Reflections on Trusting Trust”

Is Ken Thompson right? (2 min)

Is Ken Thompson right?

- Narrow answers:
  - Rewrite the compiler completely (details...)
  - Wheeler's Diverse Double Compiling (link)

Is Ken Thompson right?

- Narrow answer:
  - Rewrite the compiler completely (details...)
  - Diverse Double Compiling
- Broader answers (revealed by narrow ones):
  - Pretty much: yes. You can’t trust code that you did not **totally** create yourself (and even then, there are further problems of course)

Case study: two netizens arguing about Keepass and 1Password

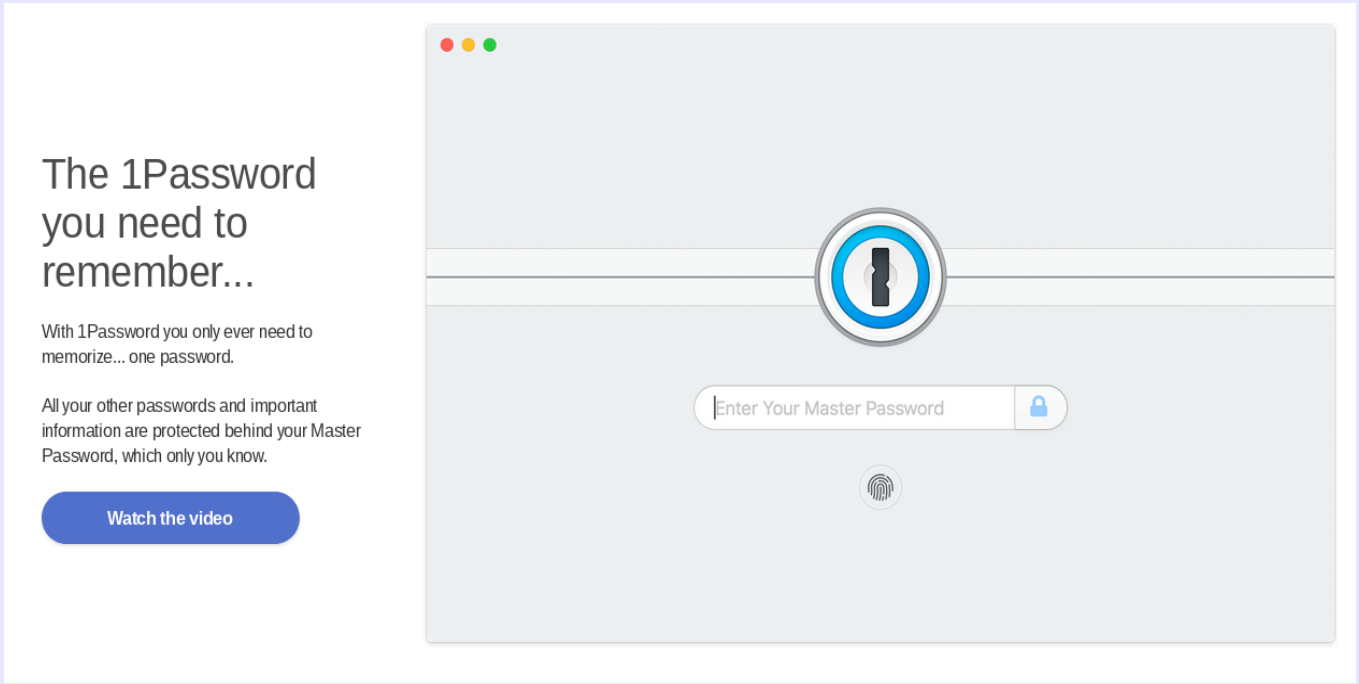
Find the snakeoil tests, and ask about whether we are stuck trusting trust

What is KeePass?
Today you need to remember many passwords. You need a password for the Windows network logon, your e-mail account, your website's FTP password, online passwords (like website member account), etc. etc. etc. The list is endless. Also, you should use different passwords for each account. Because if you use only one password everywhere and someone gets this password you have a problem... A serious problem. The thief would have access to your e-mail account, website, etc. Unimaginable.

KeePass is a free open source password manager, which helps you to manage your passwords in a secure way. You can put all your passwords in one database, which is locked with one master key or a key file. So you only have to remember one single master password or select the key file to unlock the whole database. The databases are encrypted using the best and most secure encryption algorithms currently known (AES and Twofish). For more information, see the features page.

Is it really free?
Yes, KeePass is really free, and more than that: it is open source (OSI certified). You can have a look at its full source and check whether the encryption algorithms are implemented correctly.

As a cryptography and computer security expert, I have never understood the current fuss about the open source software movement. In the cryptography world, we consider open source necessary for good security; we have for decades. Public security is always more secure than proprietary security. It's true for cryptographic algorithms, security protocols, and security source code. For us, open source isn't just a business model; it's smart engineering practice.
Bruce Schneier, Crypto-Gram 1999-09-15



I never claimed or implied that latest design of 1Pass repository is worse or even security-equivalent to KeePass. **I simply pointed out that 1Pass team has made their share of mistakes (plural), so I have as much trust in their competence as, likely, in KeePass team.**

With author trust being a non-issue (humor me in this assumption), we must look at facts & evidence only.

Both 1Pass and KeePass repositories are well-specified, with latest 1Pass clearly having an advantage due to AEAD.

1Pass implementation quality is unknown due to it being closed-source, and I'm not aware of any independent audits. KeePass implementation quality can at least be observed & discussed. 1Pass cannot even be discussed due to being a "trust-us" blackbox. Well, I don't trust them.

I would wager that even you don't know whether 1Pass actually HMAC's their IVs.

On a more holistic level, this category of software is client-based password managers (as opposed to centralized password managers like LastPass). **My position is that trustworthy client-based password managers cannot be closed-source.**

You start out in a reasonable place but then rhetorically overplay your hand: **I'm pretty sure they do HMAC their IV, (a) because they say they do and (b) because there are open source implementations of their file format that (i) do the HMAC verification and (ii) would not work properly if they weren't HMAC'ing their IV.** You can check right now: it took 2 minutes to find the Python code that computes the HMAC.

It's a minor thing to be wrong about, but it's also something you could have checked yourself before dinging me about it. :)

The story of this whole thread culminates in a place where I trust 1Password a lot more than KeePass; KeePass knows they need a better cryptosystem, but retains a broken one. 1Password has an extensively documented file format with 3rd party implementations, the author of which format actually responds to academic research.

I'd still use KeePass before I used LastPass, though, and would still use KeePass before I used no password manager!

Guilty as charged on the HMAC'ing the IV verification - bad example for a still-valid point. You still don't know whether closed-source code is using the rng properly, sending "debugging information" containing your private data to the mothership when internet is available/stars align, creating plaintext temporary files in %temp% folder (accessible by all other apps), etc, etc. Ie. there is a myriad of things the implementation could get seriously wrong, even though the repository itself is encrypted securely.

I would argue that KeePass and its loyal and vast userbase does not in fact seem to know they need a better cryptosystem (and ideally better implementation). My HN post was intended to bring this to everyone's attention.

"I'd still use KeePass before I used LastPass, though, and would still use KeePass before I used no password manager!" - so would I.

Some of the things you've mentioned here you actually can test for even on closed-source implementations. **It's pretty easy to trace the activity of the app to see if it creates temporary files or does network activity so you can you investigate that stuff.**

As for the other things, like using the rng properly and whatnot, no, you can't really check that stuff. **But your implication here is that open-source apps can be trusted because you can verify that stuff, and I don't buy that. Unless you yourself are a crypto expert that's qualified to carry out such an audit, and you have the spare time / inclination to perform a full audit of the app, then there's no reason why it being open-source should make it any more trustworthy. Perhaps if some independent trustworthy third party performed the audit you could then decide to trust it, but closed-source apps can still be audited, it just requires the help of the app developer to do so.**

...

(cont'd) ... and of course even if the app is audited (whether open- or closed-source), that audit will only really verify the particular version that was audited. **Future changes may introduce vulnerabilities again, so unless someone qualified to do so is constantly auditing all future changes, then you can't really trust it anymore, since your trust model is that the source needs to be independently verified to be trusted.**

On other hand, if your trust model is that you determine whether you trust the people involved to get it right, then it doesn't matter if the app is open- or closed-source, as long as it's developed by the right people. Granted, it can be hard to determine whether someone can be trusted to get it right without independent audits, but speaking personally, **I take tptacek's "I feel like they know what they're doing" recommendation as carrying a fair amount of weight.** I certainly would welcome an independent audit of 1Password, but I recognize that I can't really expect a closed-source software vendor to hand the source of their flagship application to a 3rd party.

(if it isn't clear, I'm a happy user of 1Password)

You can't expect a closed-source crypto software vendor to hand the source to a 3rd party, but you have no problems handing that vendor's software the keys to your life. I'm not going to debate the merits of that decision, but it's a choice you make based on your individual, hard-to-quantify perception of 'trust'.

I have ample factual evidence that both KeePass and 1Pass authors had made multiple crypto blunders. Both score low on my trustworthiness scale.

It's extremely difficult to prove crypto correct, but it's very easy to discover that it's wrong. **Open-source software allows one to discover crypto mistakes. It does not allow one to prove crypto correctness.**

On the other hand, if you use closed-source software like 1Password, you cannot discover crypto mistakes regardless of your level of crypto expertise.

...

(cont’d) ... Once we start making crypto choices based on tptacek's, schneier's, or anyone else's feelings about someone seeming to know what they are doing and getting a 'good vibe', the dark age of crypto will truly be upon us. Many folks trust & use PasswordSafe not because Schneier wrote it (I hope) but because it is open-sourced. Many folks trust & use Tarsnap not because Percival wrote it, but because the client is open-sourced.

> you have no problems handing that vendor's software the keys to your life.

I rely on a large amount of closed-source software for a great many things in my life. I'm not sure why my password manager is notably different than any other software that manages particularly important information.

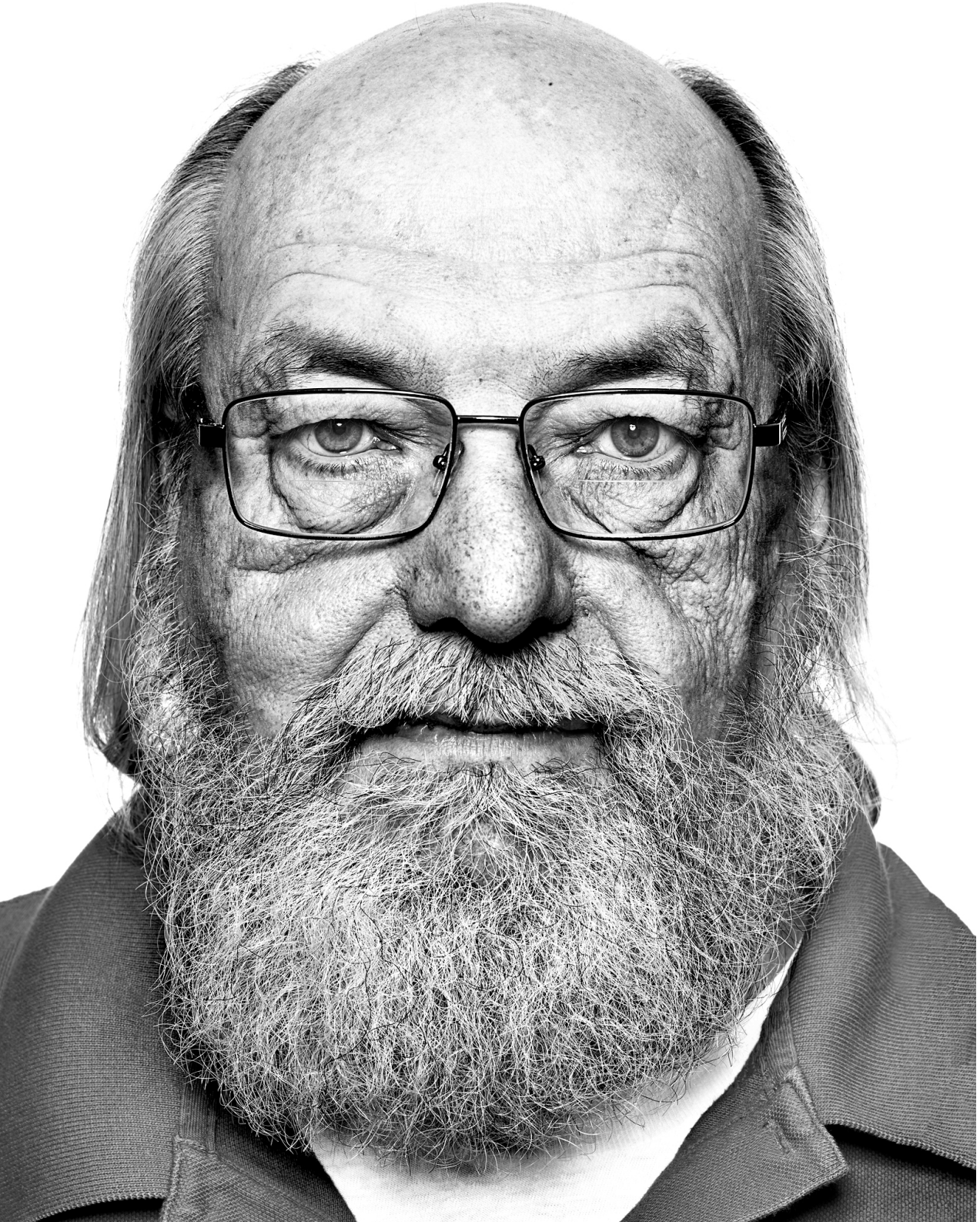
> Many folks trust & use PasswordSafe not because Schneier wrote it (I hope) but because it is open-sourced.

Virtually nobody that uses it is qualified to actually judge whether it's secure. At some point you have to put your trust in some person to tell you whether or not it's secure. In the case of a fully-audited open-source solution, you're putting your trust in the auditor to have done a good job. In the case of an open-source solution that was audited at one point but has continued development since then, you're putting your trust in a combination of the auditor to have done a good job and the original developer to have maintained the quality level of the software during subsequent development. In the case of an open-source solution that has not been audited at all, you're putting your trust in the developers, and in the anonymous collection of other people that may or may not have actually examined the source in any meaningful fashion. And in a closed-source solution, you're putting your trust in the developers.

...

(cont'd) ... The biggest problem I have with your position is you're making the implicit assumption that, just because open-source software makes its source available to the world, this means enough anonymous other people have independently audited the software in order to feel reasonably secure. But this assumption is flawed, for several reasons. **First, just because the source is available doesn't mean anyone's actually bothered to read it, and even very popular projects can suffer from this problem if the project isn't particularly accessible to contributors** (case in point, AIUI the OpenSSL source is pretty hard to grok, and historically has had very few contributors, which led to issues like Heartbleed). **Second, if people do read through the source, this doesn't in any way mean that anyone who's sufficiently qualified to judge the crypto has done so. Thirdly, even if someone who is sufficiently qualified has read through the source, it doesn't mean they've done so in a rigorous-enough fashion to really qualify as an audit.**

In the end, unless you personally are sufficiently qualified to perform an independent audit of the open-source software, and unless you personally have actually performed said audit, then you are ultimately just trusting people. Which is exactly the same situation you have with closed-source software.



The moral is obvious. **You can't trust code that you did not totally create yourself.** (Especially code from companies that employ people like me). **No amount of source-level verification or scrutiny will protect you from using untrusted code.**

Ken Thompson, ACM Turing Award Speech, “Reflections on Trusting Trust”

In my argument I never make a leap from "OSS allows discovery of crypto mistakes" to "OSS must be higher quality" or "OSS is better for the masses than closed-source".

In fact, I've never seen more crypto bs than in OSS. I'm not beating the OSS drum for the "good people of the world". OSS is a crypto requirement for me, personally, to make intelligent risk decisions.

Uneducated people have no choice but to trust someone. Educated people (ex. tptacek) should have the capability to discover crypto mistakes to make their own decisions against their own risk tolerance equation. Absence of mistakes doesn't prove anything, but their presence speaks volumes.