TCC (Tracer-Carrying Code): A Hash-based Pinpointable Traceability Tool using Copy&Paste

Katsuhiko Gondow[1], Yoshitaka Arahori[1], Koji Yamamoto[2], Masahiro Fukuyori[2] and Riyuuichi Umekawa[2]
[1]*Dept. of Computing Science, Tokyo Institute of Technology, Tokyo, Japan*
[2]*Fujitsu Laboratories, Kanagawa, Japan*

Keywords: Software Traceability, Traceability Link, Hash Value, Software Maintenance, Copy and Paste.

Abstract: In software development, it is crucially important to effectively capture, record and maintain traceability links in a lightweight way. For example, we often would like to know "what documents (rationale) a programmer referred to, to write this code fragment", which is supposed to be solved by the traceability links. Most of previous work are *retrospective* approaches based on information retrieval techniques, but they are likely to generate many false positive traceability links; i.e., their accuracy is low. Unlike retrospective approaches, this paper proposes a novel lightweight *prospective* approach, which we call TCC (tracer-carrying code). TCC uses a *hash value* as a tracer (global ID), widely used in distributed version control systems like Git. TCC automatically embeds a TCC tracer into source code as a side-effect of users' *copy&paste* operation, so users have no need to explicitly handle tracers (e.g., users have no need to copy&pastes URLs). TCC also *caches* the referred original text into Git repository. Thus, users can always view the original text later by simply clicking the tracer, even after its URL or file path is changed, or the original text is modified or removed. To show the feasibility of our TCC approach, we developed several TCC prototype systems for Emacs editor, Google Chrome browser, Chrome PDF viewer, and macOS system clipboard. We applied them to the development of a simple iPhone application, which shows a good result; our TCC is quite effective and useful to establish and maintain pinpointable traceability links in a lightweight way. Also several important findings are obtained.

1 INTRODUCTION

In software development, it is crucially important to effectively capture, record and maintain traceability links in a lightweight way. For example, we often would like to know "what documents (rationale) a programmer referred to, to write this code fragment", which is supposed to be solved by the traceability links. Most of previous work are *retrospective* approaches based on information retrieval techniques, but they are likely to generate many false positive trace links; i.e., their accuracy is low.

In practice, there are many cases where the programmers know, with *pinpoint* accuracy, the source and target of traceability links for the software artifacts being developed. For example, consider the situation where a programmer found out the URL (e.g., to Q&A sites like Stack Overflow), where a workaround is described after spending several tens of minutes to resolve an implementation issue (see Sec. 2.3 and Sec. 5 for concrete examples). We call this a *pinpointable* traceability link (Sec. 2.2).

Unlike retrospective approaches, this paper proposes a novel lightweight *prospective* approach, which we call TCC (tracer-carrying code). The main aim of TCC is to make it easier to certainly establish and maintain pinpointable traceability links, focusing on the practical use. Note that TCC does not aim to establish *all* pinpointable traceability links. Instead, TCC aims to establish *non-trivial* pinpointable traceability links, which are found out by the programmers after spending several tens of minutes for example, and thus are thought important by the programmers.

TCC uses a *hash value* as a tracer (global ID), widely used in distributed version control systems like Git. TCC automatically embeds[1] a TCC tracer into source code as a side-effect of users' *copy&paste* operation, so users have no need to explicitly handle tracers (e.g., users have no need to copy&pastes

[1]In the current implementation of TCC, the hash value is simply written in source code as comments.

URLs). The reason of using copy&paste in TCC is because our observation suggests that there is a high correlation between pinpointable traceability links and copy&paste operations. That is, a copy&paste often occurs when a programmer wants to use some information in the referred document (where to copy) to write a code fragment (where to paste), for example.

TCC also *caches* the referred original whole text into Git repository. Thus, users can always view the original text later by simply clicking the tracer, even after its URL or file path is changed, or the original text is modified or removed.

Although the idea of TCC is quite simple, TCC has the following important advantages:

- TCC automatically establishes pinpointable traceability links by hooking into users' copy&paste operations. Also TCC alleviates the bothersome management of URL links, memos, and so on. Once the user has done the TCC's copy&paste, the text stored in a Git repository can be always viewed by simply clicking the TCC tracer, even after its URL or file path is changed, or the original text is modified or removed. Furthermore, TCC can handle the artifacts with no URL or file path from the beginning.
- TCC can check the consistency among artifacts by comparing the hash values (see `pair-hash` in Sec. 4.1.2). Also TCC can cope with fine grained traceability links (see `part-hash` in Sec. 4.1.3).
- TCC is language-independent, since TCC can be applied to any text-based files. Furthermore TCC can be applied to some binary files if copy&paste is available (e.g., PDF files).
- The Git repositories for other projects can be safely merged into the Git repositories used by TCC since the possibility of the hash collisions is low enough in practice.
- Unlike retrospective approaches, TCC distinguishes different versions of the same artifact including many common words, since TCC is hash-based; different versions have different hashes in practical use.
- The TCC tracer is just a text including hash values, so even in editors or environments not supported by TCC, the contents of artifacts can be retrieved by manually manipulating the Git repository with the hash values.
- The idea of TCC is very simple, so the TCC implementation is likely to be very small. Actually the LOC of TCC for Emacs is about 1,000 lines in Emacs Lisp. (In spite of this fact, the TCC implementation is not trivial, and even not feasible in some cases. See Sec. 4.4).

The main contributions of this research are as follows:

- We have proposed TCC as a novel lightweight traceability method which automatically embeds a hash value as the tracer to the copy&pasted text (Sec. 4).
- We have provided the first prototype implementation of TCC for Emacs, Google Chrome, PDF viewer and macOS clipboard (Sec.4.3). The full source code of TCC is publicly available at the TCC homepage (TCC homepage).
- Our preliminary experiment shows that TCC is an effective and useful traceability tool with some important findings (Sec. 5).

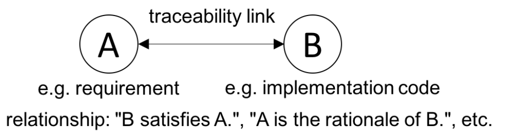
The rest of this paper is organized as follows. Sec. 2 explains the software traceability and our target pinpointable traceability links. Motivating examples of pinpointable traceability links are also presented. Sec. 3 gives related work. Sec. 4 describes the idea, design and implementation of TCC. Sec. 5 describes the preliminary experiment of applying TCC to a simple iPhone application development. Several import findings are also described. Finally Sec. 6 gives the conclusion.

2 BACKGROUND

2.1 Software Traceability

Although there are various definitions of software traceability (Cleland-Huang, et al., 2012) (Cleland-Huang, et al., 2014), we define software traceability as follows:

> *Software traceability* is a property that we can trace, in both directions, what parts of one software artifact (e.g., code) reflect (e.g. satisfy, implement, etc.) a part of another software artifact (e.g., requirement).


traceability link
A
B
e.g. requirement
e.g. implementation code
relationship: "B satisfies A.", "A is the rationale of B.", etc.

Suppose there are two software artifacts A and B. Also suppose there are a relationship between A and B like: "B satisfies A", "A is the rationale of B" and we can reach from A to B, and vice versa. Then, we say "a software traceability between A and B is established". A link between A and B is called "traceability link".

Software systems are developed to satisfy their requirements, related standards and laws, and so on, but the requirements are often modified. So we often need to modify the corresponding code, too. Conversely, when a bug is found, we need to refer to the corresponding requirements to correct the code. Therefore traceability links are indispensable in software development.

2.2 Pinpointable Traceability Links

Unfortunately, it is often quite difficult to effectively and efficiently establish traceability links. Upstream requirements are often ambiguous, vague and conflicting (Cleland-Huang, et al., 2012), and their granularity tends to be coarse, so it is difficult to connect them to the corresponding downstream code fragments. A non-functional requirement (e.g., security arrangement) often comes with cross-cutting concerns, which potentially have complicated one-to-many (or many-to-many) traceability links, and it is quite difficult to establish them in reality.

From this observation, in this paper, we limit our scope to *pinpointable* traceability links, which can pinpoint the parts of source and target of the related artifacts (see also Sec. 1). In other words, pinpointable traceability links are links with one-to-one relationship between the fine-grained parts of text-based artifacts. At a glance, pinpointable links are easy to cope with, but actually not. The reasons are as follows:

- As the size of software systems and their SDK platforms are rapidly increasing, there are so many pinpointable links among them. These APIs and documents are often updated, and their URLs are often modified. Thus, the management of pinpointable traceability links is a bothersome and error prone task.
- Even though traceability links are limited to text-based ones, there are many kinds of media like plain texts, PDF, HTML/XML documents, MS office documents, etc., which are stored in various tools (e.g., version control systems, issue tracking systems, Email clients, SNS) and IDEs (e.g., Eclipse, Xcode).
- It often takes much time (e.g., several tens of minutes) for programmers to find out pinpointable traceability links, since there are several cases where the API is used intentionally in a non-intuitive way (See Sec. 2.3 and Sec. 5 for example). One major reason of the difficulty in recovering traceability links stems from the fact that it is really difficult to precisely know the programmer's intentions, for example, when the programmer used not straightforward or not well-known ways.

```
void signal_handler (int sig) {
    // fprintf must not be used here (see C99)
    write (fd, buf, n_buf);
}
```

Figure 1: `write` is used instead of `fprintf` in a UNIX signal handler.

2.3 Motivating Examples of Pinpointable Traceability Links

2.3.1 Motivating Example #1: `fprintf` Cannot Be used in a UNIX Signal Handler

As shown in Fig. 1, in a UNIX signal handler, the system call `write` should be used to output some data, and the library function `fprintf` must not be used to do so. The reason of this is because Sec. 7.1.4.4 of the C standard (C99, 1999) says as follows:

> The functions in the standard library are not guaranteed to be reentrant and may modify objects with static storage duration. Thus, a signal handler cannot, in general, call standard library functions.

This description comes from the fact that data races can occur if the C standard I/O library functions are used in a UNIX signal handler as they (including `fprintf`) are not reentrant, for example, in the buffering process (Tahara, et al., 2008).

A traceability link from this code fragment in Fig. 1 to the above description in the C standard is obviously pinpointable one. Note that it is quite difficult for programmers to find out this pinpointable traceability link, if they do not know the possibility of data race occurrence.

Also note that it is quite difficult to establish this pinpointable link by the existing retrospective methods using the similarity between the words in the text, since the words "`fprintf`" and "signal handler" frequently appear in the other places of the C standards.

On the other hand, once we discovered this fact, we can establish the traceability link with pinpoint accuracy (thus pinpointable), since the target of the link is only one small fragment of the document (i.e., Sec. 7.1.4.4 of the C standard (C99, 1999)).

centering

```
chrome.runtime.onMessageExternal.addListener (
    function (request, sender, sendResponse) {
        chrome.tabs.query ({active: true, currentWindow: true},
                         function (tabs) { chrome.tabs.sendMessage (tabs[0].id, type: "TCC",
                                        function (res) { sendResponse (res); return true; });
                                         return true; });
        return true; // this line cannot be omitted.
    }
);
```

Figure 2: `return true;` is required to send a response asynchronously.

2.3.2 Motivating Example #2

The code fragment in Fig. 2 is a part of our TCC implementation TCC for PDF viewer (Sec. 4.3). In Fig. 2, using `chrome.runtime.onMessageExternal`, a callback function is registered to communicate between two Google Chrome extensions. The problem here is that the final response should be sent back by calling `sendResponse`, but was not, since we accidentally omitted the last `return true;` by mistake. This happened because it is not intuitive for us to tell the Chrome that we wish to send a response asynchronously by using this `return true;`, although this is clearly written in the Google Chrome manual (Google Chrome Manual) as follows.

> This function becomes invalid when the event listener returns, unless you return true from the event listener to indicate you wish to send a response asynchronously (this will keep the message channel open to the other end until sendResponse is called).

The link between the manual of `chrome.runtime.onMessageExternal` and the code fragment in Fig. 2 is another example of pinpointable traceability link. If the link is lost, it becomes difficult to know the rationale of why the last `return true;` cannot be omitted.

3 RELATED WORK

There are many papers on the techniques for establishing traceability links, but, to our knowledge, they do not use the idea of our TCC, i.e., automatically embedding a TCC tracer into artifacts as a side-effect of users' copy&paste operation.

3.1 Retrospective Traceability, Feature Location

There are many *retrospective* approaches, which try to automatically recover traceability links by calculating the similarity between the words in the artifacts, using information retrieval techniques like Latent Semantic Indexing (LSI) (Marcus, et al., 2003)(Asuncion, et al., 2010)(Dekhtyar, et al., 2007)(Delater and Paech, 2013)(Hayes, et al., 2006)(Gethers, et al., 2011). There are many feature location techniques proposed, which try to identify the source code fragment that implements some functionality (requirement), thus they are a kind of retrospective traceability approaches specific to features (Rubin and Chechik, 2013)(Dit, et al., 2011)(Dit, et al., 2013)(Ishio, et al., 2013).

Unfortunately, all of them produce only *candidates* of traceability links by similar word search, so these approaches tend to lack the accuracy and produce many false positives. Especially, they cannot distinguish different versions of the same artifact (e.g., source code, API document), while our TCC can distinguish them since different versions have different hash values in practical use.

3.2 Prospective Traceability

There are several *prospective* approaches, which try to (semi-)automatically capture traceability links by monitoring users' operations (Neumuller and Grunbacher, 2006)(Asuncion, et al., 2007)(Pinheiro and Goguen, 1996)(Pohl, 1996)(Medvidovic, et al., 2003)(Kersten and Murphy, 2005)(Altintas, et al., 2006)(Asuncion and Taylor, 2009). Unfortunately, similarly to retrospective approaches, they also suffer from the lack of accuracy. Furthermore, some approaches require users to describe some rules in advance (Pinheiro and Goguen, 1996)(Pohl, 1996)(Asuncion and Taylor, 2009). Thus, the existing prospective approaches are not lightweight.

Our TCC is a prospective approach in a broad sense, but TCC's accuracy is high and does not require rules in advance (i.e., TCC is lightweight), since TCC embeds tracers as a side-effect of users' copy&paste operation.

3.3 Software Concordance

The Software Concordance (Nguyen and Munson, 2003) integrates both of source code and documents into XML formats, and connect them using hypertext links as traceability links. Software concordance also provides uniform editing and versioning for Java. This approach seems highly language-dependent, which may make it difficult to extend the Software Concordance to other languages.

Unlike the Software Concordance, TCC is language-independent, and TCC can be in principle applied to all text-based artifacts, since TCC just embeds a hash value to the original copy&pasted text.

3.4 Documentation Tools, Cross Referencer Tools, IDEs

Documentation tools (e.g., Javadoc, Doxygen) generate API documents using the user-specified annotations, which helps users to keep the API documents consistent, but they do not support traceability links. (Users have to manually specify the URL in the annotations.)

Cross reference tools (e.g., LXR, GNU Global, Cxref) automatically generate links between the definition and reference of identifiers, which helps users to understand the source code. The supported links are limited to identifiers (e.g., names of files, classes, methods, functions and variables), so they cannot support general traceability links.

Recent IDEs (e.g., Eclipse, Xcode) navigate users to the corresponding API manuals by clicking the API function names in the source code editors. Like cross reference tools, the supported links are limited to API function names.

3.5 Literate Programming

In literate programming (Knuth, 1984), source code and the corresponding document are written in the same file, and a dedicated tool of literate programming generates the traditional source code and documents separately. This approach helps users to keep source code and its document consistent, but traceability links need to be manually described, when the document exists outside of the source code (e.g., standards and laws) or the document is too large to put them into the same file.

3.6 Software Watermarking, Software Birthmarking

There are software watermarking/birthmarking techniques (Jalil, 2009)(Myles and Collberg, 2004), whose primary goal is to prevent software piracy. In watermarking, some invisible data is embedded to programs purposely, while a birthmark is calculated using some characteristics of the programs.

Both techniques detect software piracy by comparing some characteristic values of two artifacts A and B. Thus, it is a prerequisite that you already have A and B. (If not, a brute force search is possible, but impractical.)

On the other hand, in software traceabilities, we need to reach from the artifact A to B when we have only A, and B is unknown yet. Thus software watermarking/birthmarking are not suitable for software traceabilities. Furthermore, software watermarking/birthmarking are typically used for the whole software (coarse granularity), while our TCC is used for the small fragments of artifacts (fine granularity).

4 TCC: TRACER-CARRYING CODE

This section presents the idea, design and implementation of our TCC (Tracer-Carrying Code). We propose a novel traceability method called TCC, whose primary aim is to establish pinpointable traceability links in an effective and lightweight way.

4.1 Ideas of TCC

4.1.1 Idea #1: TCC Embeds a Hash Value as Tracer in a Copy&Pasted Text

Consider the situation where a user copy&pastes some text (e.g., "`output "X"`" in Fig. 3) from one file A to another file B, with the intention of establishing a traceability link from B to A. Then, as a side effect of the copy&paste, TCC automatically embeds the hash value (0x1234 in Fig. 3) for the whole content of A to the copy&pasted text[2]. In our current implementation, we use SHA-1 (20-bytes length, i.e., 40 characters in hexadecimal) as a hashing method.

[2]TCC also embeds other information like URL if exists. See Table. 1 in Sec. 4.2 for the detail.

The pasted hash values might make the source code less readable. To avoid this problem, TCC provides a mechanism to hidden the hash values in the editors like outlining mechanism.

Also TCC registers the whole content of the file A to a Git repository, so the whole content of A can be retrieved by using the hash value as a key. Thus, the user can view the content of A by accessing the Git repository at any time, even after the original file content is modified or removed, or the URL or file path is modified. To make this retrieval easier, TCC provides a mechanism to do this by simply clicking the hash value in B's editor.

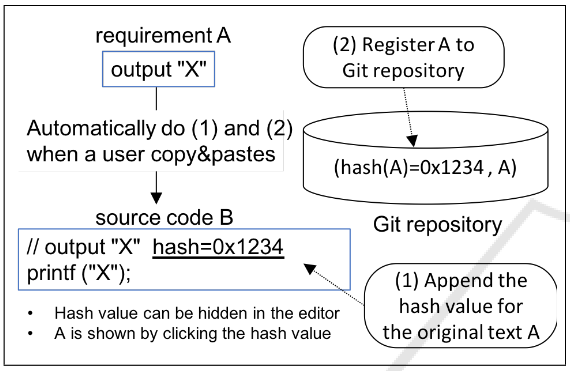

requirement A
output "X"
(2) Register A to Git repository
Automatically do (1) and (2) when a user copy&pastes
(hash(A)=0x1234 , A)
Git repository
source code B
// output "X" hash=0x1234
printf ("X");
(1) Append the hash value for the original text A
• Hash value can be hidden in the editor
• A is shown by clicking the hash value

Figure 3: TCC's idea #1.

4.1.2 Idea #2: TCC Detects Inconsistency using Pair Hash

If the user desires, TCC embeds the *pair hash* of A and B (e.g., hash(A+B) in Fig. 4), which is the hash value for the string-concatenated text of A and B. If the original file A or B (on the Internet or file system, not in the Git repository) is changed, TCC detects the change by comparing two pair hashes of the previously recorded one and the latest one.

Here this pair hash mechanism assumes that the user always sets up the pair hash after he or she modifies the file A and/or B into a consistent state between them. This assumption is required since there is no other way for TCC to know that A and B get consistent. The pair hash is a kind of declaration by the user to show the user believes that A and B are consistent.

Note that the pair hash cannot prevent malicious falsification, since any user can set up the pair hash again after modification. In this case, TCC cannot detect the inconsistency, although the previous consistent versions of A and B can be reverted manually later from the Git repository.

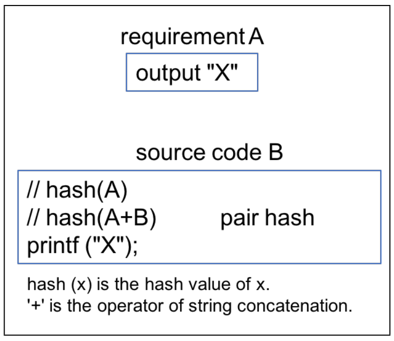


Figure 4: TCC's idea #2.

4.1.3 Idea #3: Fine-grained Traceability using Part Hash

As is often the case, the user wants to establish a traceability link not to the whole content of the file, but to the specific part of it. In this case, the hash value for the whole content might produce a false positive, since only adding one character at the beginning of the file implies the change of the whole file, even though the specific part that the user is interested in remains unchanged. Thus, the granularity of the whole file is sometimes too coarse.

To solve this problem by enabling fine-grained granularity, TCC embeds the *part hash* of the specific part that the user specifies (e.g., hash(a part A) in Fig. 5). The part hash for the file A (reference target) is straightforwardly computed for the selected region in copy&paste, while the region needs to be additionally specified by the user to compute the part hash for the file B (reference source).

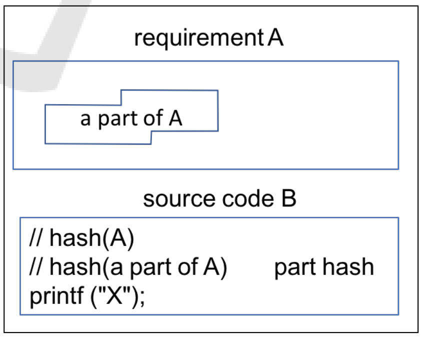
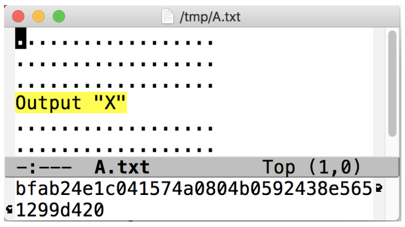


Figure 5: TCC's idea #3.

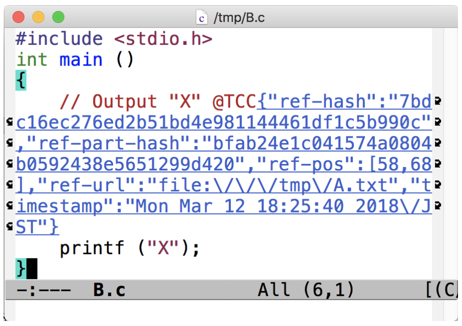
4.2 TCC's More Concrete Usage Examples

This section shows TCC's more concrete usage examples using the screenshots of the Emacs version of our TCC implementations, called TCC for Emacs



```
/tmp/A.txt
..................
..................
..................
Output "X"
..................
..................
-:---  A.txt         Top (1,0)
bfab24e1c041574a0804b0592438e565
1299d420
```

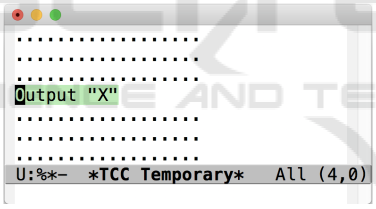
Figure 6: TCC's copy operation in the file A.txt.



```
/tmp/B.c
#include <stdio.h>
int main ()
{
    // Output "X" @TCC{"ref-hash":"7bd
c16ec276ed2b51bd4e981144461df1c5b990c"
,"ref-part-hash":"bfab24e1c041574a0804
b0592438e5651299d420","ref-pos":[58,68
],"ref-url":"file:\/\/\/tmp\/A.txt","t
imestamp":"Mon Mar 12 18:25:40 2018\/J
ST"}
    printf ("X");
}
-:---  B.c            All (6,1)      [(C
```

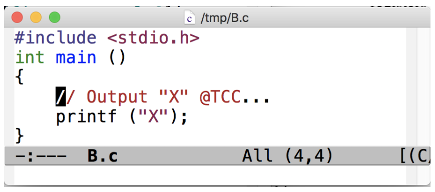
Figure 7: The copied text with a hash code is pasted into B.c.

(See Sec. 4.3 for other TCC implementations). TCC for Emacs is implemented as an Emacs minor-mode, so TCC can be enabled in editing of any kind of text-based files, which means TCC for Emacs is language-independent.



```
..................
..................
..................
Output "X"
..................
..................
..................
U:%*-  *TCC Temporary*   All (4,0)
```

Figure 8: The content of A.txt is shown with the previously copied region highlighted (in a different buffer from A.txt).



```
/tmp/B.c
#include <stdio.h>
int main ()
{
    // Output "X" @TCC...
    printf ("X");
}
-:---  B.c            All (4,4)      [(C
```

Figure 9: TCC tracers are made invisible.

1. TCC copy operation

   Consider the situation, like Sec. 4.1, where a user copy&pastes the text "output "X"" from the file A.txt to B.c. First, in the file A.txt, the user does the *copy* operation (Fig. 6). As a side effect, as described in Sec. 4.1.1, TCC appends the hash value of A.txt to the copied text. Also TCC registers A.txt to a Git repository[3]. Note here that we deliberately designed TCC's copy operation to be different from the normal one in the Emacs or the underlying OS, since TCC's copy operation silently modifies the copied text, and the modification of the copied text unintended by the user is undesirable[4]. Because of this, TCC for Emacs does not use the normal copy&paste area in Emacs (primary selection). Instead, TCC for Emacs uses the secondary selection as the TCC's copy&paste area, which most users do not use.

2. TCC paste operation

   After the copy operation, the user pastes the copied text with a hash value to B.c (Fig. 7). Unlike TCC's copy operation, this paste is a normal one. The underlined text appended by TCC for Emacs (called *TCC tracer*) in Fig. 7 is clickable; when clicked, the content of A.txt will be shown in the editor with the previously copied region highlighted (Fig. 8) by retrieving it from the Git repository. Note that the buffer window is different from the original one of A.txt, thus this feature always works even after A.txt is removed or moved in the editor, file system, or Web server.

   As shown in Fig. 7, the TCC tracer is represented in JSON format for easy processing; the keys in TCC tracer and their meaning are listed in Table. 1. @TCC is just a preamble to show that TCC's hash value follows. The TCC tracer in Fig. 7 includes the hash value of A.txt (ref-hash), the part hash value of A.txt (ref-part-hash), the byte offset positions of the copied area in A.txt (ref-pos), the file path of A.txt (ref-url), and the time stamp when copied (timestamp).

   The TCC tracer in Fig. 7 is 7-lines long; it makes the source code difficult to read. To alleviate this, TCC for Emacs allows the user to make the hash value invisible (Fig. 9) by a simple key operation (Ctrl-c t i by default).

3. TCC inconsistency checking

   TCC detects the inconsistency between source and target files by comparing the hash values in the TCC tracer. To show this, consider the situation where the user modified the first line of A.txt (Fig. 10), but the user forgot to modify the corresponding part of B.c, which results in an inconsistent state.

   Then TCC compares two hash values between

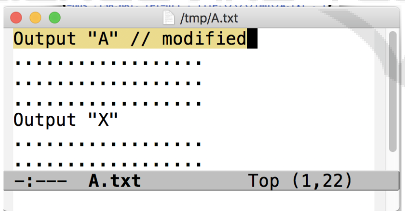
[3]We use Git in our implementation, but other hash-based distributed version control systems can be used alternatively. TCC uses the Git repository at /.tcc by default, but the existing Git repositories in other projects can also be used, since no interference occurs there.

[4]Actually, in most Web browsers, modifying the clipboard in JavaScript is not allowed for security reasons.

Table 1: Keys in TCC tracer.

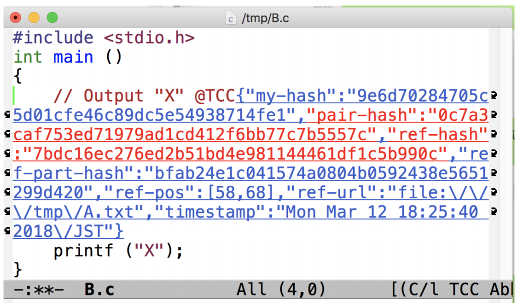
Key	Description
`ref-hash`	Hash value for the whole content of target file
`ref-part-hash`	Hash value for the part content of target file
`ref-pos`	Pair of byte offsets of the copied region in target file
`ref-url`	URL or file path of target file (if exists)
`timestamp`	Time stamp when this TCC tracer was appended
`my-hash`	Hash value for the whole content of source file
`my-part-hash`	Hash value for the part content of source file
`my-pos`	Pair of byte offsets of the specified region by the user in source file
`pair-hash`	Hash value for the whole content of both of target and source file
`pair-part-hash`	Hash value for the part contents of both of target and source file
`pair-pos`	Two pairs of byte offsets for the regions in target and source file

the previously recorded one and the latest one, and highlights them in red color if they differ (Fig. 11). Currently, this is invoked by the user's key operation (`Ctrl-c t c` by default), but it can be automated, using periodic polling, without the user's operation. Note that in Fig. 11,



```
Output "A" // modified
..................
..................
..................
Output "X"
..................
..................
-:---  A.txt           Top (1,22)
```

Figure 10: The user modifies the target file `A.txt` at the first line.



```
#include <stdio.h>
int main ()
{
    // Output "X" @TCC{"my-hash":"9e6d70284705c
5d01cfe46c89dc5e54938714fe1","pair-hash":"0c7a3
caf753ed71979ad1cd412f6bb77c7b5557c","ref-hash"
:"7bdc16ec276ed2b51bd4e981144461df1c5b990c","re
f-part-hash":"bfab24e1c041574a0804b0592438e5651
299d420","ref-pos":[58,68],"ref-url":"file:\/\/
\/tmp\/A.txt","timestamp":"Mon Mar 12 18:25:40 
2018\/JST"}
    printf ("X");
}
-:**-  B.c           All (4,0)       [(C/l TCC Ab
```

Figure 11: TCC detects the inconsistency between `A.txt` and `B.c`.

`pair-hash` and `ref-hash` are highlighted in red, but `ref-part-hash` not since the copied text `Output "X"` in `A.txt` is not modified. This demonstrates the idea of TCC's part hash works well in this example. Also note that a naive re-computation of the latest part hash using byte offsets `ref-pos` does not work, since the text `Output "X"` is shifted forward by the text insertion at the first line, and thus the hash values calculated by `ref-pos` differ from the previously recorded one, even if the text `Output "X"` is not modified. So our current implementation uses a diff tool (`git diff`); TCC reports the inconsistency if the original copied text is included in the diff.

4.3 TCC Implementations

Currently, we have implemented the following TCC prototypes, using 20-bytes length SHA-1 as a hash method, and Git as a hash-based distributed version control system:

- TCC for Emacs
- TCC for Google Chrome
- TCC for PDF viewer
- TCC for macOS clipboard

TCC for Emacs supports all the TCC operations described in Sec. 4.2. Also TCC for Emacs supports the paste operation by drag&drop, i.e., the TCC tracer is inserted by dragging a file icon and then dropping it on the Emacs editor.

The prototype implementations other than TCC for Emacs only support the TCC copy operation, since

- The Web pages and PDF files on the Internet are usually not editable[5].
- Editing local PDF files has usually no meaning, because the change will not be reflected to the original PDF files on the Internet.
- The API of macOS clipboard provides no way to edit the file where the copy operation is done.

TCC for Google Chrome is implemented as a Chrome extension. For just the same reason that TCC for Emacs uses a secondary selection (Sec. 4.2), TCC for Google Chrome does not hook the normal copy command (e.g., `Ctrl-C`). Instead, the TCC copy operation is invoked by selecting the context menu `Copy with TCC tracer` (Fig. 12) to make the user aware that the copied text is modified by adding the TCC tracer.

TCC for PDF viewer is implemented by modifying `PDF.js` [6], which is a Chrome extension. The interface

[5]This is the main reason why our current TCC only supports one way links, not bidirectional ones.

[6]https://mozilla.github.io/pdf.js/

of the TCC copy operation in TCC for PDF viewer is the same as TCC for Google Chrome (Fig. 13).

TCC for macOS clipboard is implemented as a "macOS service". The macOS services enable the user to use the features of another application, passing some data (e.g., the selected text) from one application. Thus TCC for macOS clipboard works for all the macOS applications that supports the macOS services. TCC for mac OS clipboard can be invoked by the service menu (Fig. 14), the context menu (Fig. 12 and Fig. 13) or by a system keyboard shortcut (`Ctrl-command-C` in our setting). Due to the limitations of the clipboard API, TCC for macOS clipboard only registers the copied text to the Git repository, not the whole content of the selected file (see also Sec. 4.4).

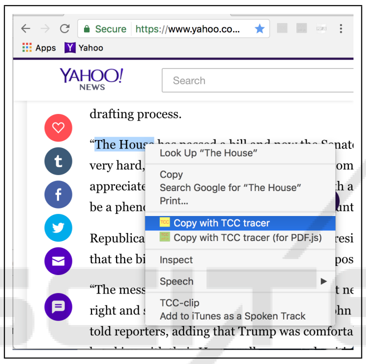

Secure https://www.yahoo.co...
Apps
Yahoo
YAHOO! NEWS
Search
Look Up "The House"
Copy
Search Google for "The House"
Print...
Copy with TCC tracer
Copy with TCC tracer (for PDF.js)
Inspect
Speech
TCC-clip
Add to iTunes as a Spoken Track

Figure 12: TCC copy operation in TCC for Google Chrome.

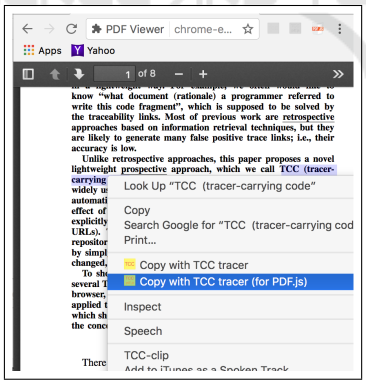

PDF Viewer chrome-e...
Apps
Yahoo
1 of 8
Look Up "TCC (tracer-carrying code"
Copy
Search Google for "TCC (tracer-carrying cod
Print...
Copy with TCC tracer
Copy with TCC tracer (for PDF.js)
Inspect
Speech
TCC-clip

Figure 13: TCC copy operation in TCC for PDF viewer.

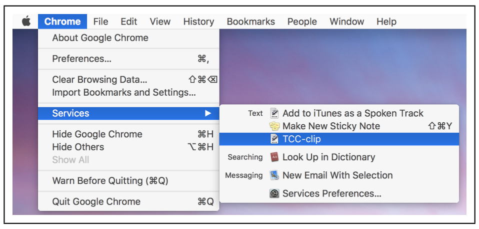


Figure 14: TCC copy operation in TCC for macOS clipboard.

4.4 The Difficulties in Implementing TCC

The idea of TCC is quite simple. Unfortunately, however, this does not imply that the TCC implementation is trivial and straightforward. Actually we encountered the following difficulties when we developed the TCC prototypes.

- The current TCC for macOS clipboard cannot compute the hash value for the whole content of the file (`ref-hash`), since the clipboard API of macOS provides no way to access to the copied file. So the current TCC for macOS clipboard only computes the hash value for the copied part (`ref-part-hash`). This shows the feasibility of implementing TCC highly depends on the underlying API.

  Also the macOS service of TCC for macOS clipboard works well for most macOS applications, but there are some exceptions (e.g., Microsoft Excel for macOS).

- There is no simple way to execute a UNIX shell command from a Chrome extension mainly due to security reasons. To work around this problem in TCC for Google Chrome and TCC for PDF viewer, we needed to implement a simple Web server (called `xhr_git_register.pl`)[7], which receives an HTML file as a POST method, executes Git commands to register the HTML file to the Git repository, and returns a SHA-1 hash value.

- There is no way to obtain the byte offset positions of the copied area (`ref-pos`) in the Chrome. The reasons are twofold.
  - The API `window.getSelection()` returns a selection object. This selection object includes relative offsets inside the selected DOM nodes, but it does not include the byte offsets from the beginning of the HTML file.
  - The HTML file obtained through the API `document.documentElement.outerHTML` and the original HTML file differs in the lexical level, where spaces, newlines and indentation are changed. Furthermore, the HTML file is normalized according to the XHTML standard, where for example `<li>aaa` is changed to `<li>aaa</li>`. This means that the byte offsets are useless even if available.

- The hash value for a file cannot be embedded in the same file correctly, in the sense that the hash

[7]Until 2013, NPAPI plugins were available, which allows us to call native binary code from JavaScript, but not available now.

```
void signal_handler (int sig)
{
    // fprintf must not be used here (see C99)
    // C99 says: a signal handler cannot,
    // in general, call (omitted) @TCC...
    write (fd, buf, n_buf);
}
```

Figure 15: TCC link is successfully established in Example. 1.

value embedded in the file is not the same as the hash value recomputed for the file, since the embedded hash value alters the hash value for the file.

To work around this problem, (e.g. `my-hash` (Table. 1) is the case), when TCC computes and checks the hash value, TCC removes the existing hash value from the file in advance.

5 PRELIMINARY EXPERIMENT

As a first preliminary experiment, we applied TCC to our two motivating examples (Sec. 2.3); for both cases, TCC worked well. Fig. 15 shows that the TCC link is successfully established in the first example (Fig. 1)[8], where the TCC link itself is made invisible as . . . .

We also applied TCC to our own simple iPhone application development. The application is a simple pedometer (Fig. 16), and its source code is about 600 lines in Swift3. As the result, we found that TCC is an effective and useful traceability tool in the following findings:

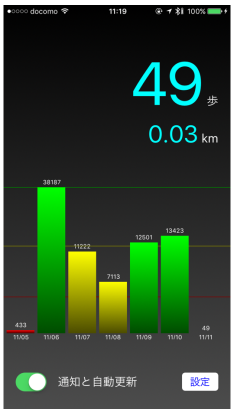


Figure 16: Screenshot of our pedometer application using TCC.

- TCC is useful to record the implementation rationale.

[8]The figure using TCC for the second example (Fig. 2) is omitted to save space here.

For example, we would like to output the notification in the application (e.g., "You have reached 10,000 steps!") even when the application is running in the background. But the motion-related API of iOS SDK (`CoreMotion`) is not allowed to run in the background. However, other famous pedometer applications like `Runtastic` definitely do this even in the background.

After a half hour struggling on the Internet, we found out some techniques in combination with other services that are allowed to run in the background (e.g., GPS, music) (Apple Programming Guide)(Apple Ref. Core)(Apple Ref. BackGround)(stack overflow). TCC made it easier for us to record these URLs as a rationale (i.e., the reason why the GPS service is used, which is unnecessary at a glance) using the TCC tracer in the source code.

This TCC tracer is useful since the TCC tracer saves time to search the rationale again, and also gives us a sense of relief that we can reach the contents of URLs at any time, even after they are updated, moved or removed (the API manuals are frequently updated).

- TCC is useful to record unresolved issues.

For example, it was unclear how we can increment an iOS badge on the application icon, which is the number of unread notifications in the application. At the time we developed, we can setup the iOS badges by incrementing the variable `UILocal.applicationIconBadgeNumber`, but the manual says that the API is deprecated, so we should not have used it.

After all, we could not find the appropriate way in the manual (perhaps, the manual was not updated for the latest API), so we could not help but use the above deprecated API. We described this issue in the Emacs buffer, and established a traceability link to this by copy&pasted from the Emacs buffer. This shows TCC is useful for unresolved issues by recording the hash value for anonymous files and stores them to the Git repository.

A typical conventional way to show some unresolved issue is to write a comment like `FIXME: this API is obsolete`. Using TCC, we can write more detailed long description and embed the TCC link to the description.

- We did not need the traceability links to most of the normal API manuals we referred to, since modern IDEs like Eclipse and Xcode provide a way to easily view the corresponding manual from the API function name. Instead, TCC is effective to record the rationale of corner cases.

6 CONCLUSION

This paper proposes a novel lightweight prospective approach to establish and maintain traceability links, which we call TCC (tracer-carrying code). TCC uses a hash value as a tracer (global ID), and TCC automatically embeds a TCC tracer into source code as a side-effect of users' copy&paste operation. To show the feasibility of our TCC approach, we developed several TCC prototype systems. We applied them to the development of a simple iPhone application, which shows a good result. As a future work, we would like to implement other prototypes, for example, for IDEs like Eclipse and Xcode, and for office software like MS Excel and Powerpoint.

REFERENCES

J. Cleland-Huang, O. Gotel and A. Zisman (Eds.): Software and Systems Traceability, ISBN: 978-1-4471-5819-6, Springer, 2012.

J. Cleland-Huang, O. Gotel, J. H. Hayes, P. Mader and A. Zisman: Software traceability: trends and future directions, Proc. on Future of Software Engineering (FOSE 2014). pp.55–69, ACM, 2014.

Programming languages–C: ISO/IEC 9899:1999.

T. Tahara, K. Gondow, S. Ohsuga: DRACULA: Detector of Data Races in Signals Handlers, 15th Asia-Pacific Software Engineering Conf. (APSEC), pp.17–24, 2008.

A. Marcus and J. I. Maletic: Recovering documentation-to- source-code traceability links using latent semantic indexing, ICSE'03, pp.125–135, 2003.

H. U. Asuncion, A. U. Asuncion and R. N. Taylor: Software traceability with topic modeling, ICSE'10, pp.95–104, 2010.

A. Dekhtyar, J. H. Hayes, S. Sundaram, A. Holbrook and O. Dekhtyar: Technique integration for requirements assessment, 15th IEEE Int. Requirements Engineering Conf. (RE 2007), pp.141–150, 2007.

A. Delater and B. Paech: Analyzing the tracing of requirements and source code during software development, a research preview, 19th Int. Working Conf. on Requirements Engineering: Foundation for Software Quality (REFSQ 2013), pp.308–314, 2013.

J. H. Hayes, A. Dekhtyar and S. K. Sundaram: Advancing candidate link generation for requirements tracing: the study of methods, IEEE Transactions on Software Engineering, 32(1), pp.4–19, 2006.

M. Gethers, R. Oliveto, D. Poshyvanyk and A. De Lucia: On integrating orthogonal information retrieval methods to improve traceability recovery, 27th IEEE Int. Conf. on Software Maintenance (ICSM), pp.25–30, 2011.

J. Rubin and M. Chechik: A survey of feature location techniques, in the book of Domain engineering: product lines, languages, and conceptual models, ISBN-10: 3642366538, Springer, pp.29–58, 2013.

B. Dit, M. Revelle, M. Gethers, D. Poshyvanyk: Feature location in source code: a taxonomy and survey, Journal of software maintenance and evolution: research and practice, 25(1), pp.53–95, 2011

B. Dit, M. Revelle and D. Poshyvanyk: Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. Empirical Software Engineering, 18(2), pp.277–309, 2013.

T. Ishio, S. Hayashi, H. Kazato and T. Oshima: On the effectiveness of accuracy of automated feature location technique, 20th Working Conf. on Reverse Engineering (WCRE), pp.381–390, 2013.

C. Neumuller and P. Grunbacher: Automating Software Traceability in Very Small Companies: A Case Study and Lessons Learne, 21st IEEE/ACM Int. Conf. on Automated Software Engineering (ASE), pp.145–156, 2006.

H. U. Asuncion, F. François, and R. N. Taylor: An end-to-end industrial software traceability tool, Proc. 6th joint meeting of the European Software Engineering Conf. and the ACM SIGSOFT Sympo. on The Foundations of Software Engineering (ESEC-FSE), pp.115–124, 2007.

F. A. C. Pinheiro and J. A. Goguen: An Object-Oriented Tool for Tracing Requirements. IEEE Softw. 13(2), pp.52–64, 1996.

K. Pohl: PRO-ART: enabling requirements pre-traceability, Proc. 2nd Int. Conf. on Requirements Engineering, pp.76–84, 1996.

N. Medvidovic, P. Grunbacher, A. Egyed and B. W. Boehm: Bridging models across the software lifecycle, Journal of Systems and Software, 68(3), pp.199âĂŞ-215, 2003.

M. Kersten and G. C. Murphy: Mylar: a degree-of-interest model for IDEs: Proc. 4th Int. Conf. on Aspect-Oriented Software Development (AOSD) pp.159–168, 2005.

I. Altintas, O. Barney, E. Jaeger-Frank: Provenance collection support in the kepler scientific workflow system, Int. Provenance and Annotation Workshop, pp.118–132, 2006.

H. U. Asuncion and R. N. Taylor: Capturing custom link semantics among heterogeneous artifacts and tools, Proc. 2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE '09), pp.1–5, 2009.

T. N. Nguyen and E. V. Munson: The software concordance: a new software document management environment, SIGDOC'03: Proc. 21st Annual Int. Conf. on Documentation, pp.198–205, 2003.

D. E. Knuth: Literate programming, Comput. J., 27(2), pp.97–111, 1984.

Jalil, Z.: A Review of Digital Watermarking Techniques for Text Documents. Int. Conf. Information and Multimedia Technology, ICIMT 2009, pp.230âĂŞ-234, 2009.

G. Myles and C. S. Collberg:Detecting software theft via whole program path birthmarks, Proc. 7th Int. Conf. on Information Security, vol. 3225 of LNCS, pages 404–415. Springer, 2004.

App Programming Guide for iOS: Background Execution, https://developer.apple. com/library/content/

documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/BackgroundExecution/ BackgroundExecution.html

Apple Reference: Core Location, https://developer. apple.com/ reference/corelocation/

Apple Reference: allowsBackgroundLocationUpdates, https://developer.apple.com/reference/corelocation/cllocationmanager/1620568-allowsbackgroundlocationupdates

stack overflow: Pedometer in the Background, http://stack overflow.com/questions/17785325/ pedometer-in-the-background/

Google Chrome Manual: chrome.runtime#onMessage External, https://developer.chrome.com/extensions/ runtime #event-onMessageExternal

TCC homepage: http://www.sde.cs.titech.ac.jp/tcc/