Exploring Large-scale System Similarity Using Incremental Clone Detection and Live Scatterplots

James R Cordy
Queen's University, Kingston, Canada
Email: cordy@cs.queensu.ca

***Abstract*—Incremental clone detection is designed to efficiently find only those clones that cross between a previous version of a system and a new version in order to update a previous clone analysis. If we instead use a different system as the new version, then it can be used to find only those clones that cross between two different systems. Live scatterplots are a visualization technique that helps localize clones quickly using pop-up information directly from points in the scatterplot. In this paper we explore how these two can be used together to rapidly expose and analyze similarities between two different systems at several levels of abstraction. Using the NiCad incremental clone detector, we find function clones between two recent versions of Linux and FreeBSD, analyze the nature and causes of some of these similarities in detail, and compare our observations with the earlier study of token-sequence clones between previous versions of these systems using CCFinder almost a decade ago.**

***Keywords*-clone detection and analysis, scatterplots**

I. Motivation and Goals

It has been almost a decade since Kamiya et al. first analyzed cloning between Linux and FreeBSD using CCFinder [19], finding, even at the fine granularity of sequences of 20 or more tokens, very little similarity between them. During that time both systems have evolved through several releases, adopted several new technologies, and grown by more than 30% in size. Both the Linux and the FreeBSD development communities say that this evolution has continued to proceed independently. It is time that we once again examined the relationship between these two large and outwardly similar systems.

Over the last decade, clone detection technology has also evolved. CCFinder is efficient and scalable in part because it works lexically, finding language-independent sequences of tokens in common. Robust parsing technology has now also evolved to the point that efficiently finding language-dependent structurally meaningful clones, such as blocks and functions, is practical at a similarly large scale using modest computers. NiCad [15] is one such parser-based clone detection method that has been shown to yield both high precision and high recall [17] in detecting near-miss intentional clones. NiCad provides a wide range of options with which to explore cloning of exact code, renamed code and structurally similar code with varying gaps quickly and easily.

Clone analysis has also evolved, and while Kamiya et al. were able to analyze the dense scatterplots generated by comparison of Linux with FreeBSD and NetBSD at both a high level and by individual example using hand analysis, it is now possible to quickly expose and analyze scatterplots automatically, using live scatterplots [5] to help uncover the nature and causes of similar subsystems.

In this first large scale experiment using analysis of cross-system cloning with live scatterplots, we pose a number of research questions.

- First, in this new world of open source and sharing, is it the case that Linux and FreeBSD have adopted any common subsystems directly from the same source, and if so, what are they?
- Second, since these systems implement a common published interface (the Posix standard for Unix-like systems), for which many algorithms have been published in the educational, academic and online literature, is there any evidence of algorithmic sharing between their functional algorithms when identifying information is abstracted away, and if so, which subsystems have such similarities?
- Third, since these are both implementations of systems with essentially the same outward interface (again, the Posix standard), is there any evidence of structural similarity between their functional algorithms when all identifying information is stripped away, and if so, to what extent?
- And finally, is there any evidence that cross-cloning between the most recent stable versions of these large systems at the whole function level is any different from that observed at the token-sequence level by Kamiya et al. almost ten years ago, or is it still the case that they exhibit a relatively low level of overall similarity?

In the remainder of this paper we explore each of these questions by varying clone detection parameters to look for each kind of similarity separately. In each case, we use live scatterplots to quickly identify the nature and apparent causes of observed similarities. We begin with a quick background on our tools: NiCad and live scatterplots.

II. Background

A. The NiCad Clone Detector

We use the NiCad incremental clone detector [6] to discover cross-system function clones between Linux and FreeBSD. The NiCad clone detection method [15] has been shown to yield both high precision and high recall [17] in detecting near-miss intentional clones at both the function and block level. NiCad is a hybrid method that combines language-sensitive parsing with language-independent similarity analysis to yield structurally meaningful near-miss clones.

NiCad involves three main stages, *parsing, normalization,* and *comparison*. In the first stage the input sources are parsed to extract all syntactic fragments of a given granularity, such as functions or blocks. Each extracted fragment ("potential clone") is pretty-printed into a standard textual form in which spacing and line breaks are normalized and comments are removed.

In the second stage, a range of plugins can be used to normalize, filter or abstract extracted fragments before comparison. In particular, potential clones can be consistently renamed to remove differences between identifiers in the same roles (e.g., replacing same identifiers by *Xn*), or "blind" renamed to remove all differences between identifiers (e.g., replacing all identifiers by *X*).

In the final stage, the extracted and normalized fragments are linewise compared using an LCS (longest common subsequence) algorithm to detect similar fragments (clones). The algorithm detects near-miss clones by allowing for linewise differences up to a given threshold, for example 0.3, corresponding to up to 30% different lines between potential clones.

The NiCad tool has two modes - whole system and incremental. In the whole system mode, it detects clones within a single version of a system or systems. In the incremental version, it takes two systems, normally a previous version and a new version of the same system, and reports only clones that cross between the versions. In this experiment we use the incremental mode in a novel way, using the most recent stable release FreeBSD (release 8.1) as the previous version of the system, and the most recent stable release of Linux (version 2.6.37) as the new version, in order to expose clones that cross between these systems.

B. Live Scatterplots

Live Scatterplots is a tool that accepts the output of any clone detector and automatically renders the clone pairs as an interactive web page that displays clones as a scatterplot over the subsystems (directory structure) of the original source system or systems. The axes are specified as two files of directory names to be used for the rows and columns of the plot respectively. These files specify the level of abstraction to be used. They can be source file names (for small systems), low level source directories (for mid-sized systems) or higher-level subsystem directories (for very large systems). The axes can use the same set of directory names (if the plot is of clones in a single system) or two different sets (for clones between two systems). In this experiment, the Y axis is the directory structure of the sys/ subdirectory of the FreeBSD 8.1 source, and the X axis is the directory structure of the linux/ subdirectory of the Linux 2.6.37 source.

Live Scatterplots constructs the scatterplot on these axes as an HTML table, using cell colour to represent clone density, where reds represent a high density of clones, greens fewer, and blues only one or two. It can display either the entire matrix, or can reduce the axes to include only those directories which actually contain clones. In either case, rendering is fast and accurate in modern web browsers.

The thing that makes Live Scatterplots "live" is its rendering of the scatterplot with pop-up titles associated with every cell. By simply hovering the mouse over any particular area in the scatterplot, the browser automatically displays the pop-up cell title, giving us immediate detailed information about the corresponding subsystems or clones. This information is exactly what we are looking for when we pose the question "what is that?", and will help us to quickly understand the nature of the observed cloning between Linux and FreeBSD.

C. The Systems

In the CCFinder experiment comparing FreeBSD, NetBSD and Linux for token-sequence clones a decade ago, versions 4.0 of FreeBSD and 2.4.0 of Linux were compared with NetBSD 1.5. Since that time, four major revisions (5.0, 6.0, 7.0 and 8.0) and more than twenty releases of FreeBSD have been released, and its kernel source has grown by a factor of three, from 1,015,619 C source lines in 4.0 to 2,946,686 lines in the current stable release, 8.1. In the meantime the Linux kernel has gone through one major revision (2.6) and fifteen releases, growing by a factor of four, from 2,366,001 C source lines in 2.4.0 to 10,275,891 lines in the current stable release, 2.6.37. In this study we analyze C function cross-cloning between the two most recent stable releases, FreeBSD 8.1 (about 3 million lines) and Linux 2.6.37 (about 10 million lines).

III. Experiment 1: Evidence of common adoption

In the intervening years between the 2002 CCFinder study and today, the growth of the open source movement has made a wide range of code publicly available for direct use. In particular, critical industry standards such as encryption and security algorithms have been developed and released as open source, and it is common practice for drivers for new hardware to be released as open source. In this environment, it is also the case that programmers more

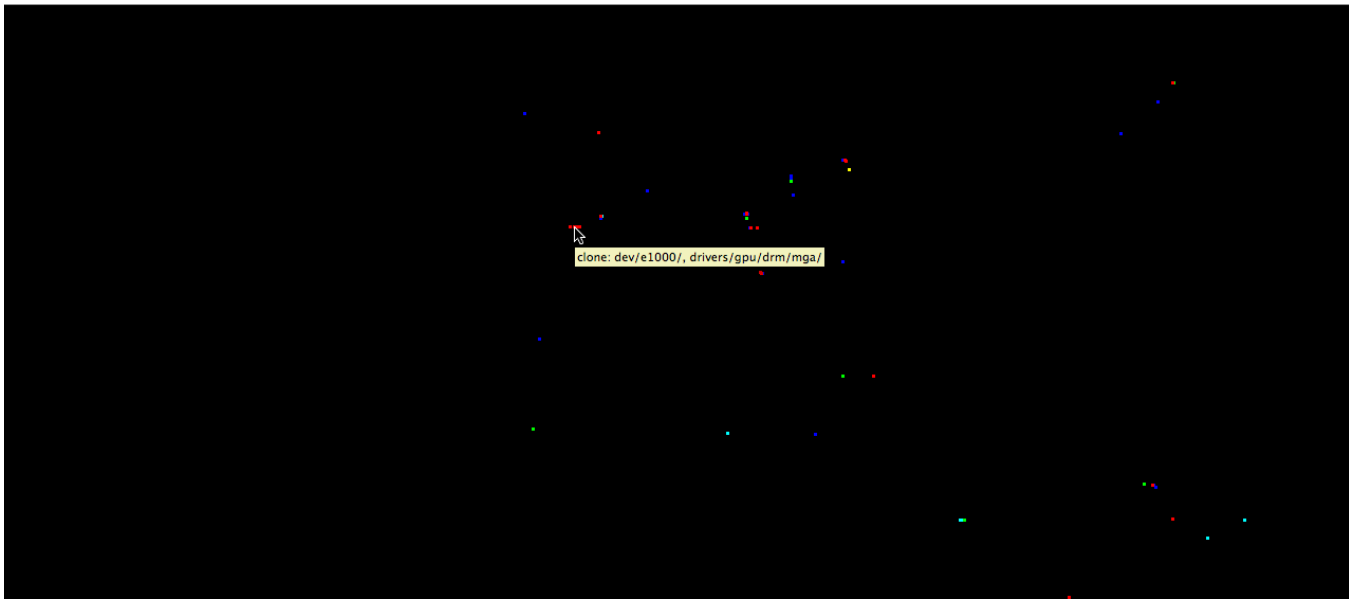


Figure 1. Raw Live Scatterplot of Linux (X axis) vs FreeBSD (Y axis) exact function clones at the 0.3 near-miss threshold

The very small number of clone points is rendered on a black background with colour cells magnified to ten times normal relative size to make them visible at this scale. Red cells indicate significant levels of cross-cloning. Rendered in the Safari 5 web browser.

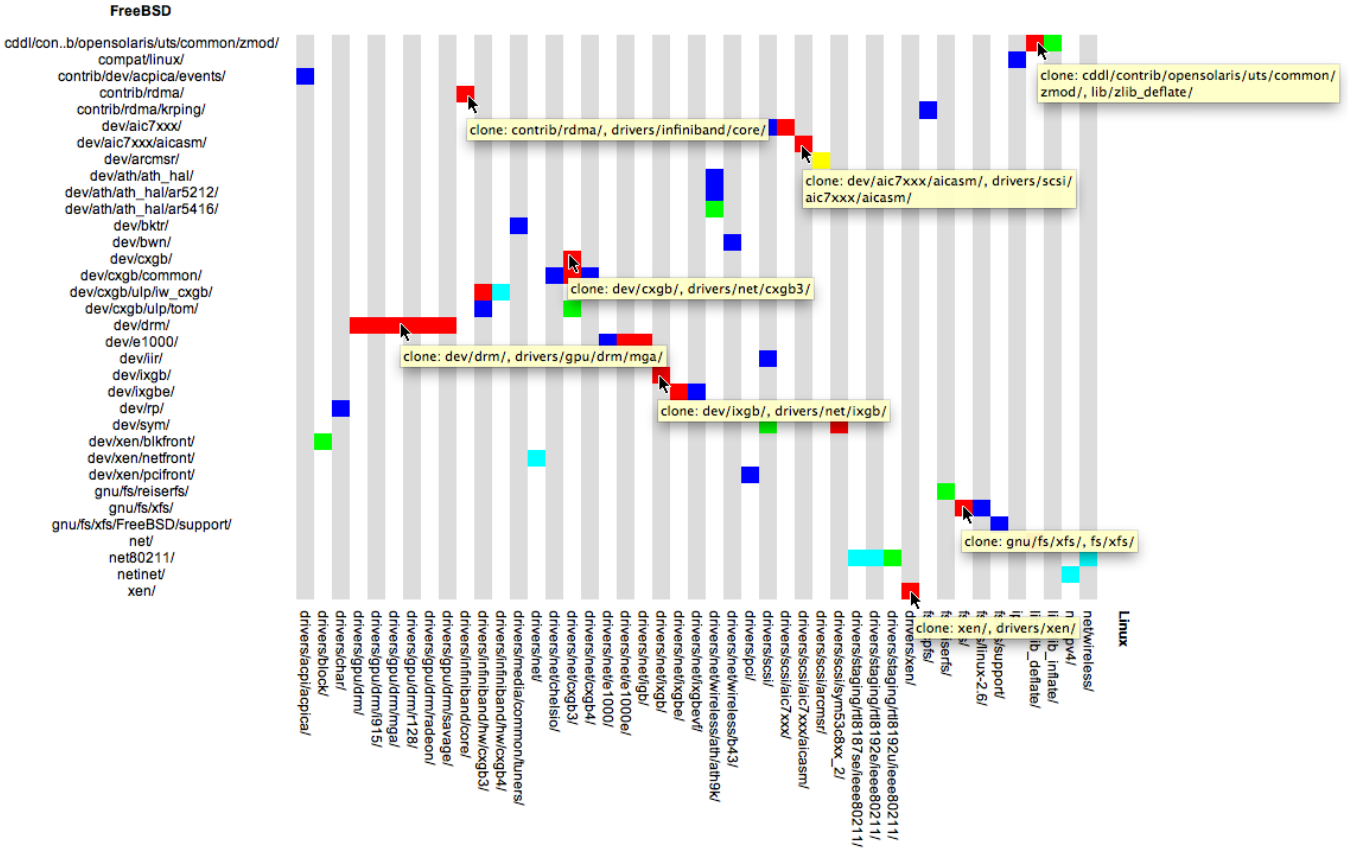


Figure 2. Axis-reduced Live Scatterplot of Linux (X axis) vs FreeBSD (Y axis) exact function clones at the 0.3 near-miss threshold

Live scatterplot pop-ups for hovering over various interesting areas of the plot are shown. (This is a composite over time, only the pop-up under the mouse pointer appears at any given time.) Rendered in the Safari 5 web browser.

frequently search the web for existing code solutions when adding new features or interfaces to their code.

It is therefore interesting to ask the question, "have these systems adopted, traded or borrowed outright any common code over the years?" In the modern open source scenario, we might expect that code such as implementations of industry standards might be directly copied, perhaps with minor changes, into the systems. To test whether FreeBSD and Linux have done any such common adoption, in our first experiment we simply tested whether any code has been copied verbatim into or between the two systems. This can be tested using exact clone detection, with a near-miss threshold to account for gaps due to minor changes.

Figure 1 shows the results of using NiCad incremental clone detection to find exact function clones between FreeBSD 8.1 and Linux 2.6.37 at a near-miss threshold of 30%. The first thing we can observe is that the global answer to the question is "no". There is essentially no exact common code at the function level shared between the current Linux and FreeBSD stable releases. Indeed, on this global scale showing all source directories, there are virtually no visible near-miss exact function clones at all unless the few clone points that do exist are magnified to ten times their relative size in the live scatterplot.

The second answer to the question is "yes", there is at least some evidence of common copied code between these systems. NiCad reports that 1,261 of the 244,336 functions in Linux (0.516%) have near-miss exact copies in FreeBSD. In Figure 2, the scatterplot has been automatically axis-reduced to only those few source directories that actually do contain clones. We can see that the common functions there are between these systems are very highly localized – there are a small number of regions of dense exact cloning (indicated by red), and several single function clones (indicated by green and blue) scattered elsewhere.

By hovering our mouse over the red patches, the live scatterplot reveals the nature of the subsystems that contain the densest groups of exactly common functions (Figure 2). Not surprisingly, almost all of the common adopted code is in the device driver subsystems of the two systems. We can see that they share copied DRM code (the wide red bar on the left), and have adopted the same driver code for rdmi (Infiniband remote direct access memory), cxgb (the Chelsio 10 Gb ethernet controller), ixgb (the Intel 10Gb ethernet controller), aix7xxx (the Adaptec family of disk controllers), zlib (an open source compression library), xfs (the SGI XFS file system), and xen, the industry standard Xen virtualization hypervisor. Live scatterplots make it quick and easy to find out what these are (and using the same browser that is displaying the scatterplot, we can Google their names to yield the more detailed information about what they mean).

IV. Experiment 2: Evidence of algorithmic similarity

If there is very little literally copied common code between the systems, perhaps it is the case that common code templates or published open-source algorithms have been customized to each environment by adapting names to the particular environment in which is it used in each system. The question we are asking in this case is "is there evidence that these systems have adopted or borrowed common code templates or standard algorithms if we allow for programmer adaptation of names?"

In this next experiment, we reexamine the systems with consistent renaming to see if perhaps there is a higher level of similarity between these operating systems from an algorithmic point of view. For this purpose we use consistent renaming as a rough approximation to algorithmic abstraction. Consistent renaming retains the original role of all identifiers, but renames them for comparison to standard names of the form *Xn* for each occurrence of each same identifier. This preserves the overall algorithm of the code fragment, but removes simple naming differences in the copies.

As in Kamiya et al.'s 2002 CCFinder experiment, we normalize single statements in *if, do, else, for* and *while* statements to compound blocks to correct for differences in bracketing style, using a NiCad normalization plugin. Other normalizations described by Kamiya et al., such as removal of initialization lists, are possible in NiCad, but unnecessary since they are already handled by a combination of pretty-printing (e.g., initializations are pretty-printed to one long line rather than many) and near-miss matching.

Figure 3 shows a raw scatterplot of the cross-system function clone pairs reported by incremental NiCad with normalization and consistent renaming, again at the 0.3 near-miss threshold level to allow for minor changes in up to 30% of the individual lines. Surprisingly, while the distribution of near-miss algorithmic cross-clones is much wider than exact clones across the systems, there are still very few to report: only 2,621 functions of Linux (1.07%) appear in FreeBSD in exact or consistently renamed form at a near-miss threshold of 0.3. This seems far less that one might expect if programmers were routinely cut-and-pasting public example algorithms, so again our first answer to this question is basically "no".

The distribution of common consistently renamed functions between the two systems is however consistent with the hypothesis that programmers are reusing common examples in their work. The more or less evenly distributed occurrence of one or two common functions (the blue, green and cyan points in the right hand two thirds of the live scatterplot of Figure 3) across the systems certainly indicates some such use. However, one can ask why there are few on the left hand side, and so few in the top tenth of the plot. As it

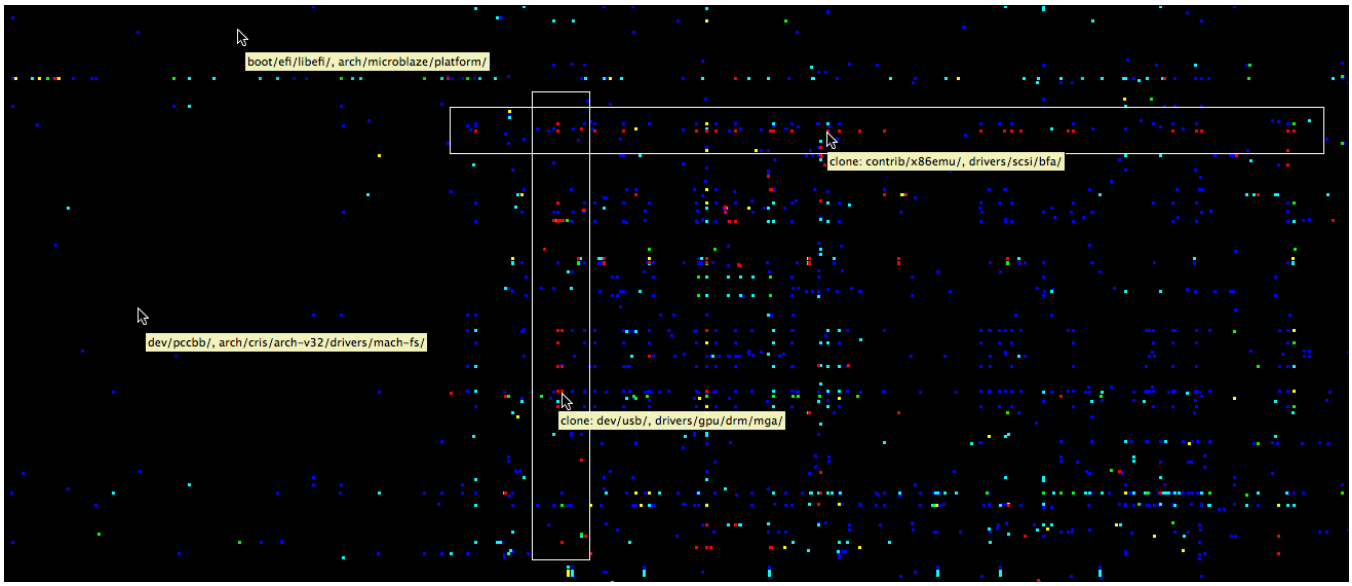


Figure 3. Raw Live Scatterplot of Linux (X axis) vs FreeBSD (Y axis) consistently renamed function clones at the 0.3 near-miss threshold

Rendered with black background and colour cells magnified to ten times normal relative size. While many interesting patterns are visible, two particular regions with significant amounts of dense cross-cloning are highlighted. Rendered in Safari 5.

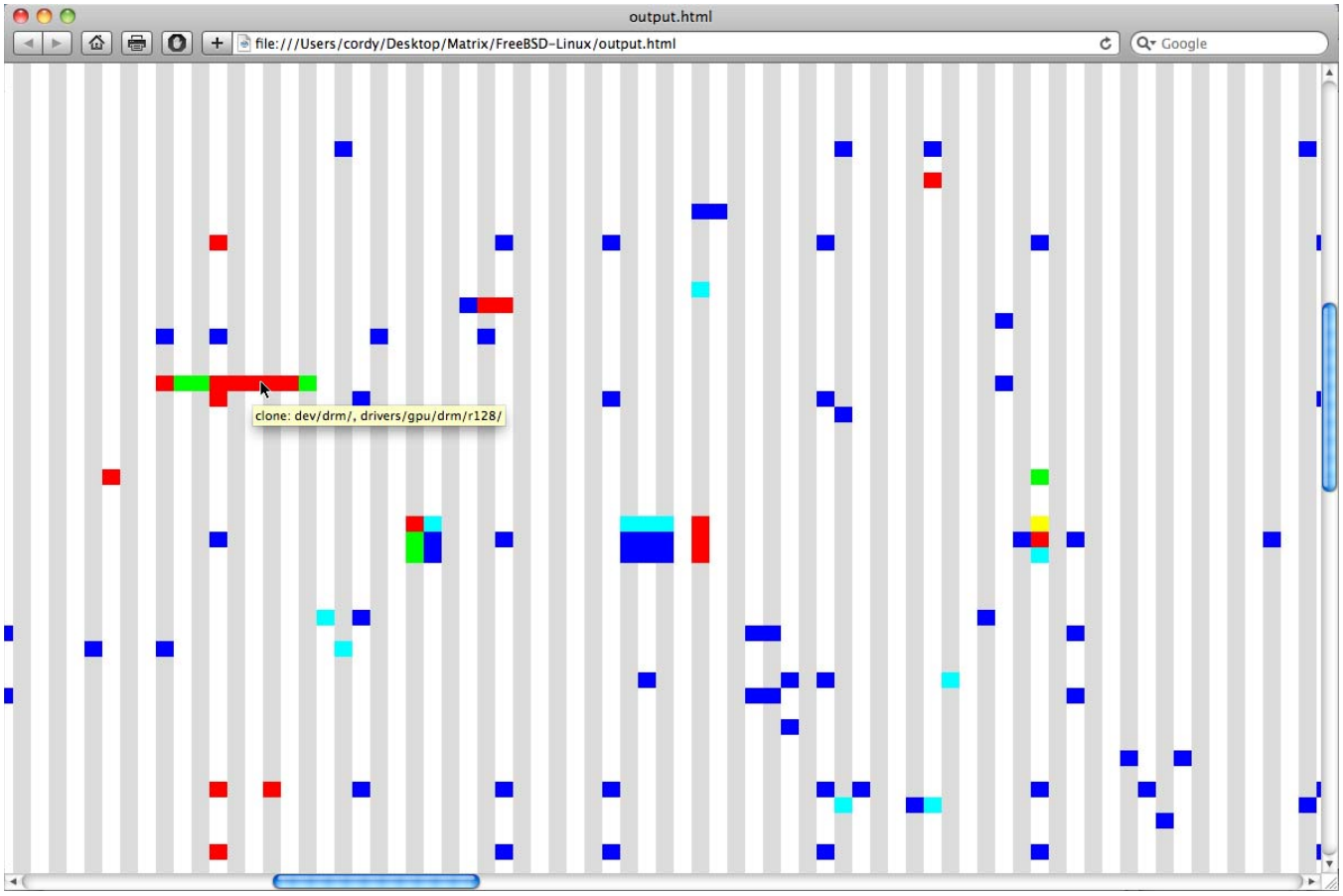


Figure 4. Axis-reduced Live Scatterplot of Linux (X axis) vs FreeBSD (Y axis) consistently renamed function clones at the 0.3 near-miss threshold

Although the complete live scatterplot is too large to look at all at once, browser scrolling and live popups still allow for detailed interactive analysis. Rendered in Safari 5.

turns out, by hovering over those areas we discover that these are the directories of Linux and FreeBSD associated with platform dependent low level bootstrap code (arch/ in Linux and boot/ in FreeBSD), which would be unlikely to be copied from any common code.

Two quite distinct patterns of dense common functions, indicated by clusters of red points, one horizontal (indicating spread across Linux) and one vertical (spread across FreeBSD) are quite visible in Figure 3 and are highlighted. Investigating the first (horizontal) one reveals that it is associated with Linux functions in common with the x86emu subsystem of FreeBSD (emulation code ported to FreeBSD and Linux from X11, according to Google), and the second (vertical) one turns out to be from the DRM code required for displaying protected video using various graphics cards, which obviously both systems require. An interesting thing is that because of the different source code organization strategies, the first of these is localized in FreeBSD but spread all over the Linux repository, whereas the second is localized in Linux but spread over FreeBSD's source code directories.

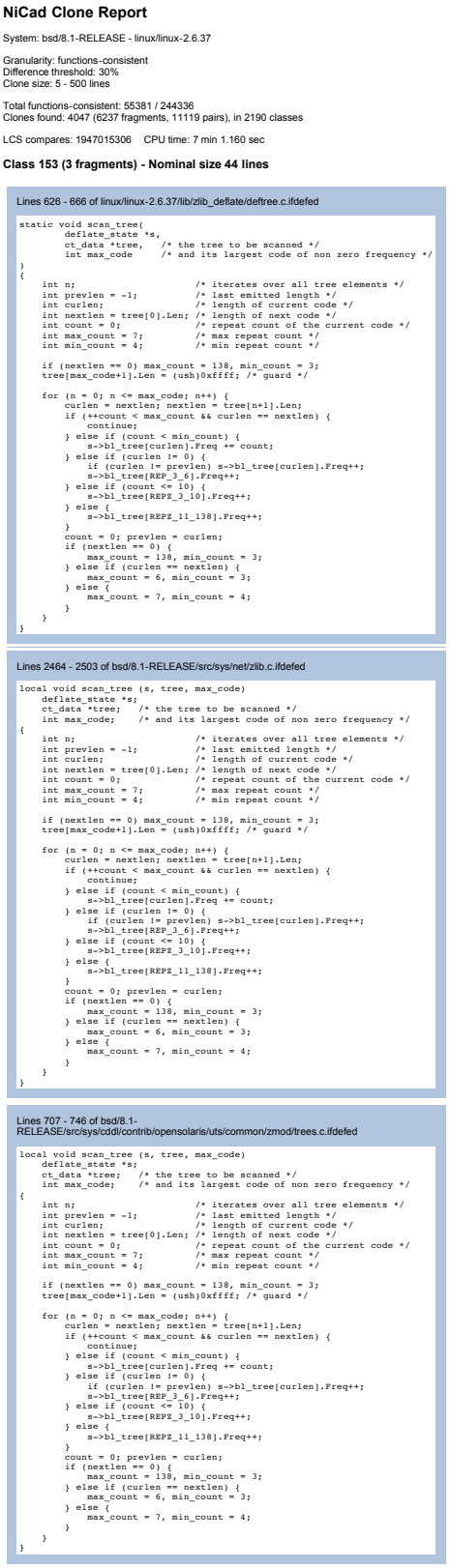
There are other interesting patterns to explore in Figure 3. For example the wide line of low-level cloning from one subsystem of FreeBSD near the top of the plot that seems to be cloned widely across all of Linux. Hovering over the pattern tells us that all of these functions are in the cddl/contrib/opensolaris/uts/common/ subsystem of FreeBSD, which a quick web search reveals is the Open Solaris kernel (UTS = UNIX Time Sharing). In fact these turn out not to be one thing, but rather a range of similarities in functions that both systems have in common with Open Solaris, including open source compression (zlib) algorithms, ioctl (disk driver) code, and numerous small (5-10 line) simple functions. Figure 5 shows the NiCad incremental clone class report for one of the larger clones associated with this commonality.

As clone density increases, the scatterplot for larger systems becomes very large, even in the case of the axis-reduced and labelled version (Figure 4). An advantage of working with a browser-based visualization is that it automatically adopts to this situation for us, by providing scroll bars for both directions, allowing us to explore interesting patterns in more detail.

There are other smaller patterns visible in Figure 3, but given the low level of overall algorithmic cloning they involve it would be misleading to spend much time on them.

V. Experiment 3: Evidence of structural similarity

Since both of these systems implement essentially the same outward interface, the Posix standard for Unix-like systems, even if they have not adopted code or algorithms from common sources into the systems, one might still expect that the algorithms used to implement the systems



NiCad Clone Report

System: bsd/8.1-RELEASE - linux/linux-2.6.37

Granularity: functions-consistent
Difference threshold: 30%
Clone size: 5 - 500 lines

Total functions-consistent: 55381 / 244336
Clones found: 4047 (6237 fragments, 11119 pairs), in 2190 classes

LCS compares: 1947015306 CPU time: 7 min 1.160 sec

Class 153 (3 fragments) - Nominal size 44 lines

Lines 626 - 666 of linux/linux-2.6.37/lib/zlib_deflate/deftree.c.ifdefed

```
static void scan_tree(
        deflate_state *s,
        ct_data *tree,   /* the tree to be scanned */
        int max_code     /* and its largest code of non zero frequency */
)
{
    int n;                      /* iterates over all tree elements */
    int prevlen = -1;           /* last emitted length */
    int curlen;                 /* length of current code */
    int nextlen = tree[0].Len; /* length of next code */
    int count = 0;              /* repeat count of the current code */
    int max_count = 7;          /* max repeat count */
    int min_count = 4;          /* min repeat count */

    if (nextlen == 0) max_count = 138, min_count = 3;
    tree[max_code+1].Len = (ush)0xffff; /* guard */

    for (n = 0; n <= max_code; n++) {
        curlen = nextlen; nextlen = tree[n+1].Len;
        if (++count < max_count && curlen == nextlen) {
            continue;
        } else if (count < min_count) {
            s->bl_tree[curlen].Freq += count;
        } else if (curlen != 0) {
            if (curlen != prevlen) s->bl_tree[curlen].Freq++;
            s->bl_tree[REP_3_6].Freq++;
        } else if (count <= 10) {
            s->bl_tree[REPZ_3_10].Freq++;
        } else {
            s->bl_tree[REPZ_11_138].Freq++;
        }
        count = 0; prevlen = curlen;
        if (nextlen == 0) {
            max_count = 138, min_count = 3;
        } else if (curlen == nextlen) {
            max_count = 6, min_count = 3;
        } else {
            max_count = 7, min_count = 4;
        }
    }
}
```

Lines 2464 - 2503 of bsd/8.1-RELEASE/src/sys/net/zlib.c.ifdefed

```
local void scan_tree (s, tree, max_code)
    deflate_state *s;
    ct_data *tree;   /* the tree to be scanned */
    int max_code;    /* and its largest code of non zero frequency */
{
    int n;                      /* iterates over all tree elements */
    int prevlen = -1;           /* last emitted length */
    int curlen;                 /* length of current code */
    int nextlen = tree[0].Len; /* length of next code */
    int count = 0;              /* repeat count of the current code */
    int max_count = 7;          /* max repeat count */
    int min_count = 4;          /* min repeat count */

    if (nextlen == 0) max_count = 138, min_count = 3;
    tree[max_code+1].Len = (ush)0xffff; /* guard */

    for (n = 0; n <= max_code; n++) {
        curlen = nextlen; nextlen = tree[n+1].Len;
        if (++count < max_count && curlen == nextlen) {
            continue;
        } else if (count < min_count) {
            s->bl_tree[curlen].Freq += count;
        } else if (curlen != 0) {
            if (curlen != prevlen) s->bl_tree[curlen].Freq++;
            s->bl_tree[REP_3_6].Freq++;
        } else if (count <= 10) {
            s->bl_tree[REPZ_3_10].Freq++;
        } else {
            s->bl_tree[REPZ_11_138].Freq++;
        }
        count = 0; prevlen = curlen;
        if (nextlen == 0) {
            max_count = 138, min_count = 3;
        } else if (curlen == nextlen) {
            max_count = 6, min_count = 3;
        } else {
            max_count = 7, min_count = 4;
        }
    }
}
```

Lines 707 - 746 of bsd/8.1-RELEASE/src/sys/cddl/contrib/opensolaris/uts/common/zmod/trees.c.ifdefed

```
local void scan_tree (s, tree, max_code)
    deflate_state *s;
    ct_data *tree;   /* the tree to be scanned */
    int max_code;    /* and its largest code of non zero frequency */
{
    int n;                      /* iterates over all tree elements */
    int prevlen = -1;           /* last emitted length */
    int curlen;                 /* length of current code */
    int nextlen = tree[0].Len; /* length of next code */
    int count = 0;              /* repeat count of the current code */
    int max_count = 7;          /* max repeat count */
    int min_count = 4;          /* min repeat count */

    if (nextlen == 0) max_count = 138, min_count = 3;
    tree[max_code+1].Len = (ush)0xffff; /* guard */

    for (n = 0; n <= max_code; n++) {
        curlen = nextlen; nextlen = tree[n+1].Len;
        if (++count < max_count && curlen == nextlen) {
            continue;
        } else if (count < min_count) {
            s->bl_tree[curlen].Freq += count;
        } else if (curlen != 0) {
            if (curlen != prevlen) s->bl_tree[curlen].Freq++;
            s->bl_tree[REP_3_6].Freq++;
        } else if (count <= 10) {
            s->bl_tree[REPZ_3_10].Freq++;
        } else {
            s->bl_tree[REPZ_11_138].Freq++;
        }
        count = 0; prevlen = curlen;
        if (nextlen == 0) {
            max_count = 138, min_count = 3;
        } else if (curlen == nextlen) {
            max_count = 6, min_count = 3;
        } else {
            max_count = 7, min_count = 4;
        }
    }
}
```

Figure 5. Linux function in common with two functions in FreeBSD

The first function is from Linux, the other two are clones of it found in FreeBSD. Rendered in Safari 5.

might bear some similarity to one another. In this case the question to be asked is, "is there any evidence of structural similarity in the functional code used to implement these outwardly similar systems?".

For this experiment, we loosened the clone definition by asking NiCad to use blind renaming (i.e., every identifier is mapped to *X*), again with normalization of consistent compound blocks for *if, do, else, for* and *while*, and with a 30% near-miss threshold. Figure 6 shows the result of running NiCad to find cross-system clones using these more aggressive settings. The first thing to notice is that there is much more visible evidence of widespread cloning between the systems, although still at an overall low level (the amount of colour evident in the live scatterplot is actually deceptive since colour points have been enhanced to ten times their real size for visibility in printed form). When naming is ignored and bracketing is normalized, at the 30% near-miss level NiCad reports that 6,123 of the 244,336 functions in Linux (2.5%) are similar to at least one function in FreeBSD, and conversely 3,059 of the 55,385 functions in FreeBSD (5.5%) have at least one clone in Linux.

While there are many interesting features to explore in the scatterplot in Figure 6, three features draw particular attention: the thin vertical line highlighted in the center of the plot, and the two thin horizontal lines near the bottom of the plot. The vertical pattern indicates a set of function structures in common with a particular subsystem of Linux that are spread widely across the FreeBSD source directories, whereas the horizontal patterns indicate function structures in common with particular subsystems of FreeBSD that are spread widely across the Linux source. Each of these patterns has many red cells, indicating a high level of clone density.

Using the live scatterplot we can again do some detailed exploration of which subsystems are involved in these patterns by hovering over them (Figure 6). In the case of the vertical pattern, all of the structurally similar functions are localized to the drivers/net/appletalk/ subsystem of Linux, the implementation of Apple's legacy AppleTalk networking protocol. In FreeBSD, the structurally similar functions to those in this subsystem are broadly spread, but almost all in the dev/ subsystem, across various device drivers. This is interesting in that it is possibly evidence of different localization criteria used in the source organizations of the two systems, with the AppleTalk protocol logic isolated into a subdirectory in Linux, and similar logic scattered among the device drivers of FreeBSD.

Hovering over the uppermost of the two horizontal lines near the bottom of the plot reveals that these are clustered around the isa/ subsystem of FreeBSD, the legacy Industry Standard Architecture (ISA) bus driver logic and associated devices. As we can see, the structurally similar function logic in Linux is spread widely over the system, mostly in the drivers/ and kernel/ subsystems. This may again be an indication of differences in source code organization between the two systems. The lower horizontal line of similar functions is much less dense (i.e., less red) but still a clear feature of the plot. Hovering over it reveals that it is mostly associated with the net80211/ subdirectory of the FreeBSD source, which houses the 802.11 (wireless networking) logic of FreeBSD, and again we see that structurally similar functions appear throughout the drivers/ and kernel/ subsystems of Linux, possibly indicating that networking logic is spread more widely in the Linux source code structure.

VI. Experiment 4: Function vs. token-sequence overall similarity

In our final experiment, we are interested to know if the observed level of function clones between the current stable releases of Linux and FreeBSD is consistent with the low level of cross-system token-sequence clones first observed by Kamiya et al. in the much smaller earlier versions of the systems in 2002. There are really two questions here, although they are difficult to separate without revisiting the entire Kamiya experiment: "are the levels of function clones observed between these systems consistent with the levels of token-sequence clones previously reported?", and "has the level of cross-cloning between these systems changed significantly over the past decade of system source growth?".

The most similar clone definition we have used to that defined by Kamiya et al. is the one in Experiment 2 (Section IV above), with consistent naming and bracket normalization similar to that used in the CCFinder experiment. As in Kamiya's 2002 experiment, name differences are simplified, and *if, do, else, for* and *while* statements are normalized to compound blocks to correct for differences in bracketing style. Other normalizations described by Kamiya, such as removal of initialization lists, are handled by a combination of pretty-printing (e.g., initializations are pretty-printed to one long line) and near-miss matching. The minimum size of cross-system clones in Experiment 2 is set to five pretty-printed lines, roughly equivalent to the 20 normalized tokens used as the minimum in the 2002 CCFinder experiment.

To compare these results directly with Kamiya et al.'s study, we must count clone pairs rather than cloned functions, since their results report on the basis of clone pair coverage. However, NiCad reports 4,984 clone pairs between functions in Linux and FreeBSD at the 30% near-miss level with consistent renaming and normalized bracketing, significantly higher than the 1,091 clone pairs reported by Kamiya, which given the relative growth of the systems by a factor of three (for FreeBSD) to four (for Linux) over the intervening decade, might predict a current level of about 3.5 x 1,091 = 3,818 clone pairs, assuming cloning levels have remained consistent over the years. Since CCFinder was detecting token-sequence clones in any context, not just those limited to whole functions, we might actually have expected to see a higher level of cloning than that reported

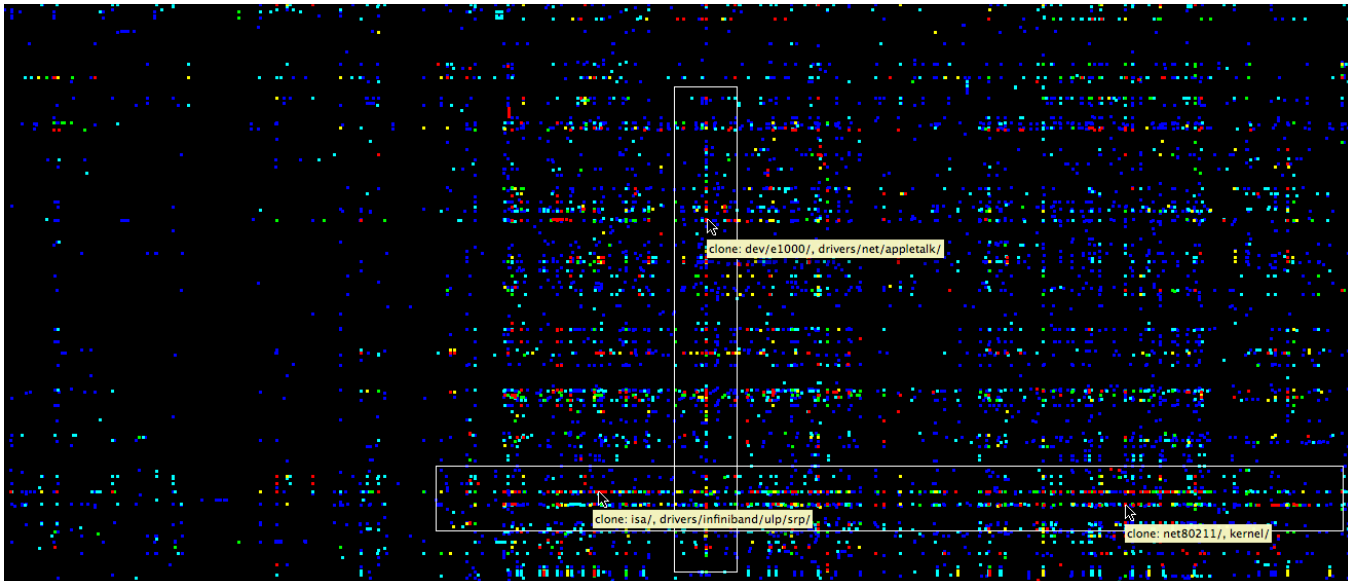


Figure 6. Raw Live Scatterplot of Linux (X axis) vs FreeBSD (Y axis) blind renamed and normalized function clones at the 0.3 near-miss threshold

Rendered on a black background with colour cells magnified to ten times normal relative size to make them more visible at this scale. While many interesting features are visible, three particular regions with broadly distributed cross-cloning are highlighted.

by NiCad at the function granularity rather than lower. This may be an indication that cloning in these systems has in fact increased with the continued growth of the open-source movement.

The CCFinder main results were reported in terms of file and line coverage of the clone pairs, and we can also compare on that basis. Table I shows the results of converting the function clone pairs observed in Experiment 2 to the numbers of original source code lines and source files that they cover in each system. As we can see, the observed coverage of source files by function clones detected in our experiment is significantly higher than the coverage originally observed by Kamiya et al., by a factor of about four in the case of FreeBSD and two in the case of Linux. This is consistent with our hypothesis above that cross-cloning between the systems has increased over the years.

In the case of lines of code coverage, however, we see a different picture. While the proportion of lines of source code of FreeBSD covered by our function cross-clones in Experiment 2 is about double the proportion covered by those reported by CCFinder in 2002, the proportion of lines of code in Linux covered by the function cross-clones is only about half of that previously observed for token sequences. While it's not completely clear what this may indicate, it is definitely the case that the majority of function cross-clones detected are at ten lines or fewer, so many small clones are involved in the results. Since Linux is such a very large system, this fact may simply be more visible at the source lines level than at the overall function or file level in Linux than in FreeBSD.

While it is not a similar clone definition, if we were to compare to the more aggressive structural similarity results of Experiment 3 (Section V) to the original CCFinder results, the number and proportion of files and lines covered by the reported function clones is overwhelmingly larger than that reported in 2002, covering more than 27% of the files and 9% of the source lines in FreeBSD, and 20% of the files and 2.5% of the lines in Linux.

VII. Threats to Validity

There are a number of assumptions we have made in this study. In Experiment 1, we modelled the borrowing or sharing of common code as near-miss exact clones (those that are identical when comments and spacing is ignored, except for small percentage of lines, in this case up to 30%). While we have high confidence that such code has been copied from a common source, we do not know for certain.

In Experiment 2, we modelled the use of published examples, templates and algorithms as near-miss consistently renamed clones at function granularity. We have assumed that a near-miss threshold of 30% allows for adaptation of the algorithm to its environment; this may be too little (or too much). We have also assumed that such copied algorithms normally appear as a whole function, which is not necessarily the case.

Experiment 3 uses near-miss blind renamed function clones as a surrogate for structural similarity of code. While this seems a valid model, it does report many trivial cases where the content of the function is simply a sequence of calls to other functions, without any real logic involved. It

	FreeBSD		Linux	
	CCFinder 2002	**NiCad 2011**	**CCFinder 2002**	**NiCad 2011**
Granularity	>= 20 token sequence	>= 5 line function	>= 20 token sequence	>= 5 line function
Files	3.1%	14.03%	4.6%	8.45%
Source lines	0.8%	1.76%	0.9%	0.56%

Table I
COMPARISON OF OBSERVED FREEBSD / LINUX CROSS-SYSTEM CLONE COVERAGE BY SOURCE FILES AND SOURCE LINES USING CCFINDER [19] (TOKEN SEQUENCES, 2002) AND NICAD [15] (FUNCTIONS, 2011)

might have been better to filter these out since they have no real internal structure.

Experiment 4 is attempting to compare apples with oranges, and makes a number of assumptions. Because we are intentionally comparing a non-structural technique with a syntactic structure technique, the parameters are quite different and the methods are looking for clones of quite different granularities. We have tried to mitigate this by choosing the most comparable settings for NiCad, with roughly the same normalization as CCFinder, roughly the same minimum clone size, roughly the same near-miss threshold for gaps, and the same strategy of not reporting clones embedded in larger clones.

Differences in clone granularity (token sequence vs. function) and gap granularity (tokens vs. lines) can of course be expected to give quite different results. Nevertheless, the reported results showing a larger percentage of cloned code across the two systems using a more restrictive (structural) clone definition give a strong indication that there is currently a significantly greater proportion of cloned code between FreeBSD and Linux than there was in 2002.

VIII. RELATED WORK

While the original Kamiya et al. study [19] is the best known cross-system clone analysis for Linux and FreeBSD, many other studies of cloning in these systems have been done over the years. Yu et al. [20] have since studied cross-system clones between Linux versions 2.1.114 and 2.4.20 vs. FreeBSD 3.0 and 5.0 respectively, concentrating on the co-evolution of clones. Li et al. [12] studied both Linux and FreeBSD using CP-Miner, relating bugs to copy-paste cloning in each, although they did not look at cross-system clones. German et al. [7] recently studied copy-paste cross-cloning between Linux, FreeBSD and OpenBSD, analyzing the legal and technical implications.

There are too many individual clone studies of each of Linux and FreeBSD to mention. One of the earliest studies of cloning in the Linux kernel was by Godfrey et al. [8] who observed that Linux was growing at a super-linear rate. Based on the relation between its size then (2 million lines) and its size now (over 10 million lines), it seems that is still true. Antoniol et al. [2] used a metrics-based method to analyze Linux release 2.4.0 for duplicated code, concluding as we have again observed that the proportion of code affected is not large. They extended their analysis to clone evolution shortly thereafter [3], concluding that the level of cloning was relatively stable over several releases (2.4.0 - 2.4.18), which is not what we have observed over a longer term. Livieri et al. [14] revisited clone evolution in Linux in 2007 using a distributed version of CCFinder (D-CCFinder [13]) and heat maps to visualize clone coverage. Chang et al. [4] used multiple versions of FreeBSD as a target for their recent evaluation of code clone detection methods.

Roy and Cordy [16] have previously used NiCad to study function and block clones in a range of open source systems, including Linux 2.6.24. Jiang and Hassan [10] used Linux versions 1.0 to 2.6.16 as the target of their data mining-based framework for clone understanding, and demonstrated that the level of internal cloning in Linux has grown enormously over the years. Deckard [9] is another parser-based clone detection method that has been used to analyze clones in the Linux kernel, using Euclidean vectors to characterize parse subtrees for comparison, whereas NiCad uses pretty-printed text lines. Kapser and Godfrey [11] have related token-sequence clones and function clones directly using their CICS visualizer to help categorize clones detected by CCFinder, and demonstrated their work on the file subsystem of Linux.

In terms of other kinds of cross-system studies, Livieri et al. also used D-CCFinder [13] to conduct an extensive cross-system clone analysis of many systems, including FreeBSD and Linux, to yield clone scatterplots across many systems at once. Al-Ekram et al. [1] undertook an extensive study of cross-cloning across a range of editors and window manager systems written in C, observing (as we also observed for Linux and FreeBSD in this paper) that "true" copies between systems seem to be unusual, whereas similar algorithms ("accidental clones") are much more common.

This study is to our knowledge the first to use a parser-based language-sensitive method to study cross-system clones, and the only study to compare results for cross-system near-miss clones at all three of the copy-paste, algorithmic and structural levels. It is the first to use live scatterplots to assist in detailed analysis of clone sources, and far as we are aware, the first to revisit a previous cross-clone experiment after a decade of evolution of the systems involved.

IX. Conclusions

In this work we have revisited the cross-cloning relationship between Linux and FreeBSD almost a decade after the original CCFinder study. In the interim these systems have grown in source code size by a factor of three to four times. Using incremental clone detection to efficiently find cross-system clones, we looked for evidence of three different kinds of cloning activity: cut-and-paste direct use of unchanged common code, common reuse of published algorithms or code templates, and structural similarity of functional code. In each case we used live scatterplots to quickly explore the nature of the similarities and expose possible sources. Finally, we compared our parser-based function clone results with the original token-sequence clone results reported by Kamiya et al. in 2002, and observed some evidence of a higher level of cross-cloning in the current stable versions of Linux and FreeBSD than previously observed. In our continuing work we are using incremental clone detection and live scatterplots in other ways, including exploring the evolution of functions in large scale systems.

Acknowledgements

This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and by an IBM Centre for Advanced Studies faculty award.

References

[1] R. Al-Ekram, C. Kapser, R.C. Holt and M.W. Godfrey. "Cloning by accident: an empirical study of source code cloning across software systems", In *Intl. Symp. on Empirical Software Engineering*, pp. 376-385, Noosa Heads, Australia, Nov. 2005.

[2] G. Antoniol, U. Villano, M. Di Penta, G. Casazza and E. Merlo, "Identifying Clones in the Linux Kernel", In *Proc. Intl. Workshop on Source Code Analysis and Manipulation*, pp. 92-99, Florence, Italy, 2001.

[3] G. Antoniol, U. Villano, E. Merlo, M. Di Penta, "Analyzing cloning evolution in the Linux kernel", *Information and Software Technology* 44:13, pp. 755-765, 2002.

[4] H.-F. Chang and A. Mockus, "Evaluation of source code copy detection methods on FreeBSD", In *Proc. Intl. Working Conf. on Mining Software Repositories'*, pp. 61-66, Leipzig, Germany, 2008.

[5] J.R. Cordy, "Live Scatterplots", Tool demonstration in *Proc. ICSE Fifth Intl. Workshop on Software Clones*, to appear, Waikiki, Hawaii, 2011.

[6] J.R. Cordy and C.K. Roy, "The NiCad Clone Detector", Tool demonstration submitted to *Intl. Conf. on Program Comprehension*, Kingston, Canada, 2011.

[7] D.M. German, M. Di Penta, Y.-G. Gueheneuc and G. Antoniol, "Code siblings: Technical and legal implications of copying code between applications", In *Proc. 6th Intl. Working Conf. on Mining Software Repositories*, pp.81-90, Vancouver, Canada, May 2009.

[8] M.W. Godfrey and Q. Tu, "Evolution in Open Source Software: A Case Study", In *Proc. IEEE Intl. Conf. on Software Maintenance, pp. 131-142, San José*, California, 2000.

[9] L. Jiang, G. Misherghi, Z. Su and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones", In *Proc. Intl. Conf. on Software Eng.*, pp. 96-105, 2007.

[10] Z.M. Jiang and A. Hassan, "A framework for studying clones in large scale software systems", In *Proc. Intl. Workshop on Source Code Analysis and Manipulation*, pp. 203-212, Paris, France, 2007.

[11] C. Kapser and M.W. Godfrey, "Aiding comprehension of cloning through categorization", In *Intl. Workshop on Principles of Software Evolution*, pp. 85-94, Kyoto, Japan, 2004.

[12] Z. Li, S. Lu, S. Myagmar and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code", *IEEE Trans. Softw. Eng.* 32:3, pp. 176-192, March 2006.

[13] S. Livieri, Y. Higo, M. Matushita and K. Inoue, "Very-Large Scale Code Clone Analysis and Visualization of Open Source", In *Proc. Intl. Conf. on Software Eng.*, pp. 106-115, 2007.

[14] S. Livieri, Y. Higo, M. Matsushita and K. Inoue, "Analysis of the Linux Kernel Evolution Using Code Clone Coverage", In *Proc. Intl. Workshop on Mining Software Repositories*, pp. 22-1-22-4, 2007.

[15] C.K. Roy and J.R. Cordy, "NICAD: Accurate Detection of Near-miss Intentional Clones using flexible pretty-printing and code normalization", In *Proc. Intl. Conf. on Program Comprehension*, pp. 172-181, Amsterdam, Netherlands, June 2008.

[16] C.K. Roy and J.R. Cordy, "An Empirical Study of Function Clones in Open Source Software", In *Proc. Working Conf. on Reverse Engineering*, pp. 81-90, Antwerp, Belgium, 2008.

[17] C.K. Roy and J.R. Cordy, "A mutation / injection-based automatic framework for evaluating code clone detection tools", In *Proc. Mutation 2009*, pp. 157-166, Denver, USA, April 2009.

[18] C.K Roy, J.R. Cordy and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach", *Sci. Comput. Program.* 74:7, pp. 470-495, May 2009.

[19] T. Kamiya, S. Kusumoto and K. Inoue. "CCFinder: a multilinguistic token-based code clone detection system for large scale source code", *IEEE Trans. Softw. Eng.* 28:7, 654-670, July 2002.

[20] L. Yu, S. Ramaswamy and J. Bush, "Symbiosis and Software Evolvability", *IT Professional* 10:4, pp. 56-62, July-Aug. 2008.