2011 IEEE Symposium on Security and Privacy

PRISM: Program Replication and Integration for Seamless MILS

Chris Owen, Duncan Grove, Tristan Newby, Alex Murray, Chris North and Michael Pope
C3I Division, Defence Science and Technology Organisation, Edinburgh, Australia
Email: {Chris.Owen, Duncan.Grove, Tristan.Newby, Alex.Murray, Chris.North, Michael.Pope}@dsto.defence.gov.au

***Abstract*—We describe how to combine a minimal Trusted Computing Base (TCB) with polyinstantiated and slightly augmented Commercial Off The Shelf (COTS) software programs in separate Single Level Secure (SLS) partitions to create Multi Level Secure (MLS) applications. These MLS applications can coordinate fine grained (intra-document) Bell LaPadula (BLP) [6] separation between information at multiple security levels. The untrusted COTS programs in the SLS partitions send at-level file edits as *diff* transactions to the TCB. The TCB *verifies* that BLP semantics will be observed and then *patches* these transactions into its canonical representation of the file. Finally, it releases appropriately filtered versions back to each SLS partition for re-assembly into the COTS program's standard file format. Furthermore, by judiciously restricting how the user can interact with the system the multiple SLS instantiations of the COTS program can be made to appear as if they are a single MLS instantiation. We demonstrate the utility of this approach using Microsoft Word and DokuWiki.**

***Keywords*-Computer security, Data security, Information security, Multilevel systems, Software architecture, Application virtualization, Military computing, Data storage systems, File systems, Information entropy**

I. Introduction

Traditional Multi-Level Secure (MLS) information systems have labelled individual files with the highest security level of information they contain, or relied on MLS databases. The latter requires the entire database engine to be trusted, which is well beyond the state of the art to prove with any degree of formal credibility. The former precludes trusted, fine grained (intra-document) labelling of content.

For example, Starlight [3] enforces strictly separate enclaves for information at different security levels but then provides mechanisms to access that information in the isolated domains. Although highly secure, this stymies users from combining information from different classifications, thus discouraging fine grained information management and inducing a fragmented user experience. The MYSEA architecture [14] demonstrates a more integrated albeit much less trustworthy approach using a wiki environment, but the granularity for different security levels is still limited to a page; a commercial MLS operating system is used to apply a mandatory separation policy using security labelled processes and files. Still other approaches like Compartmented Mode Workstation (CMW) [5], [18] apply a high-water mark model and "float" files up to the highest security level of other processes that touch them. Unfortunately this over classification quickly ratchets everything to the highest level, constraining the availability of data and forcing users into frustrating and risky upgrade/downgrade cycles.

Unlike these approaches, the system which we present in this paper, PRISM, uses a minimal cross domain component to provide trustworthy separation of security levels while maintaining cross domain structure *within* individual files, providing much finer-grained partitioning of MLS data. In this respect our approach is similar to Galois' Trusted Services Engine (TSE) [12], although little detail has been published about this system which makes it difficult to understand exactly how it works and determine its useability and security properties.

Notwithstanding this, the TSE is described as implementing a trusted "read down" mechanism whereby low-level data can be accessed from higher-level domains. A high side user can then make high-level modifications to the data, presumably using some form of copy-on-write semantics. Separate TCB-mediated disks are used to store the information at different security levels. Additional untrusted "DocServer" software on the high side monitors the low side file for modifications and merges those changes onto the high-level version of the document – although no details of how this is achieved are provided.

Using this approach, high-level edits that alter low-level content (but within the high-level domain) are implicitly allowed. This is clearly demonstrated in the TSE's main articulated use case, where high-level users are allowed to edit both low- and high-level wiki content from the same high-level workstation. This in turn drives the need to introduce a fallible downgrading process into a user's typical workflow, where ostensibly low-level information entered at a high security level needs to be examined, filtered and resynchronised downwards to the low level.

Conversely, our PRISM system provides the user with a simple mechanism for always editing MLS content *at the appropriate security level*, thereby avoiding the need for risky downgrade procedures. Furthermore, PRISM generates fine grained edit transactions and "pushes" them, through a TCB, to higher levels. In addition to providing the user with the apparent synchronisation of data between security levels, frequent, fine grained edit transactions help PRISM avoid merge conflicts that are inevitable when using an occasional heavyweight diffing process. Our interposed TCB is also able to provide additional value adding steps. In particular, it maintains a trusted, canonical MLS version of the file. This assists with the trusted labelling and tracking of MLS information, which in turn simplifies release checking procedures, and also allows our MLS documents to be cryptographically sealed and exported for off-line use.

Section II lays out the architecture of our solution. Section III explains the underlying MLS file format and Section IV describes the trusted processing that it requires. For completeness, Section V briefly presents the cross-domain infrastructure that we relied on to develop our prototypes. Sections VI and VIII demonstrate how we augmented Microsoft Word and DokuWiki to make them compatible with our approach. Section VII shows how we built upon these platforms to deliver a seamless user experience. Section IX analyses the security properties of our solution, including an analysis of potential covert channels and susceptibility to other threats. We conclude with possible avenues for further work.

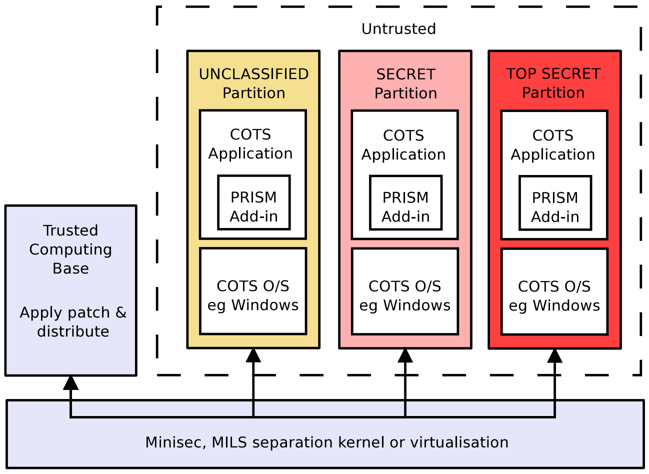


Figure 1. The PRISM Architecture.

II. The PRISM Architecture

The Multiple Independent Levels of Security (MILS) architecture [2], [23] provides a robust framework for separating untrusted code (typically Commercial Off The Shelf (COTS) software) from trusted, security-critical code. This enables complex yet highly assured applications to be constructed by placing the large and complex parts of applications in untrusted system-high partitions, while security critical functions are separated out into small, trusted components. This minimises the Trusted Computing Base (TCB) comprising the security critical functionality and greatly eases evaluation and certification effort. For example, it substantially reduces the amount of formal mathematical modelling required to achieve Common Criteria (CC) certifications of Evaluation Assurance Level (EAL) 6+ and above, transforming an intractable problem to a manageable one.

The Annex Minisec device (versions 2 and 3) – on which we aim to publish details soon [10] – applies the MILS concept at both coarse and fine grained levels. At the macroscopic level, a Minisec is composed of multiple self-contained computing systems (including CPU, RAM, storage, etc.) for running complete system-high COTS operating systems and associated software stacks. Each of these self-contained systems – called a *partition* – is typically associated with a distinct security level, and may also form part of a wider network at that level – called an *enclave*. At the microscopic level the Minisec's dedicated Trusted Computing Base (TCB) CPU runs an Annex Object Capability Reference Monitor (OCRM), which is an updated version of the Annex TCB described in [9] that we will describe more fully in [13]. Layered on top of this is a suite of objects implementing Multi-Level Security critical functions that, in conjunction with the Minisec's labelled networking mechanism, dynamically mediate each partition's access to: (1) local I/O devices such as a screen, keyboard, mouse and audio; (2) partitioned networking services for communicating with corresponding partitions in the same enclave on remote devices; and (3) between partitions (of inherently different security levels) on the same device. This paper focuses on interactions of the third kind.

As shown in Figure 1, our PRISM architecture uses Program Replication and Integration for Seamless MILS (PRISM). It combines a minimal TCB with polyinstantiated and augmented COTS *programs* (rather than data) in separate Single-Level Secure (SLS) partitions to create MLS applications. The untrusted COTS applications in the separate system-high partitions are augmented with untrusted PRISM *add-in* modules that facilitate communication and synchronisation with trusted components in the TCB, allowing coordinated fine grained (intra-document) editing of MLS documents with Bell LaPadula (BLP) [6] separation by single or multiple users operating at multiple security levels.

This fusion of COTS software running in MILS domains under the policy control of a true MLS TCB delivers the best of all worlds: the power of COTS, the security properties of MILS and the usability of native MLS. While this style of composition was clearly anticipated in early MILS publications [2], [23], the modern concept of MILS appears to have degenerated to Multiple SLS (MSLS) type systems aimed at consolidating what are currently "air-gapped" guest operating systems. We suggest that our fine grained PRISM architecture is returning to MILS' original roots, but propose that "Multiple *Integrated* Levels of Security" might better explain this type of architecture.

Figure 2 shows how an untrusted COTS *Application* in the SLS partitions creates or edits a file. An untrusted but MLS-aware *Differ* applies delta compression [11] to translate any at-level modifications into diff transactions. These are communicated to the TCB where a tiny MLS *Verifier* trivially checks whether BLP security semantics will be preserved by

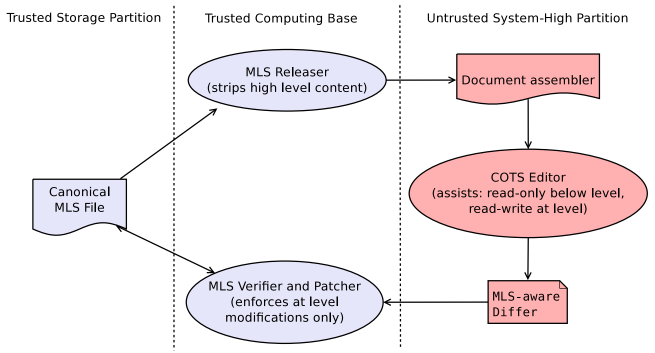

Trusted Storage Partition
Trusted Computing Base
Untrusted System-High Partition
MLS Releaser (strips high level content)
Document assembler
Canonical MLS File
COTS Editor (assists: read-only below level, read-write at level)
MLS Verifier and Patcher (enforces at level modifications only)
MLS-aware Differ

Figure 2. Editing information with PRISM.

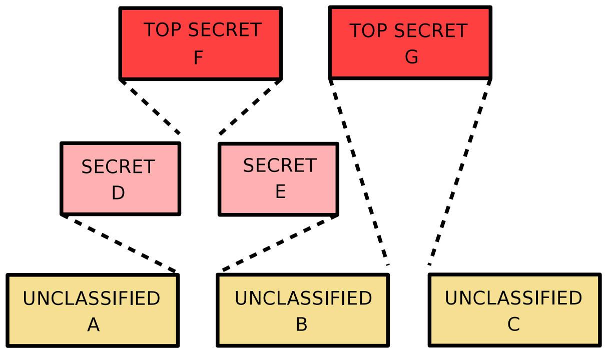


Figure 3. Conceptual file structure

the modifications, and if so, the *Patcher* merges the changes into the *canonical version* of the file. The canonical version of the file may be maintained within the TCB partition, stored in a dedicated MLS storage partition, or hosted in what we term *system-low* mode as a TCB-encrypted file on one of the untrusted partitions. A tiny MLS *Releaser* then creates sanitised versions of the canonical file at each security level, trivially stripping any higher-level content. Each of these sanitised versions is communicated back to the appropriate SLS partition, where an untrusted *Assembler* uses the filtered canonical representation to reconstruct the file into the COTS application's standard file format.

This enables trusted separation of information at different security levels without needing to trust the COTS applications or the operating system partitions in which they reside. Our key insight was to combine program replication with a delta compression interface in a MILS framework to separate the untrusted editing application(s) from a minimal, trustworthy update and coordination mechanism.

Furthermore, all data is permanently classified at the security level of the edit session that created it. However, by making data easily accessible from any security level dominating partition, the need to upgrade and downgrade information is completely avoided in everyday use. Instead, upgrade and downgrade operations become rare, necessary only when explicitly re-grading data to a different level.

III. The MLSDoc File Format

Our canonical container file format for storing MLS data, called MLSDoc, prescribes a sequence of objects, each of which is simply a byte sequence at a particular security level. Where consecutive objects at the same classification occur – due to at-level additions, or deletions of intervening higher-level content – these become merged into a single object. Consequently, each data byte of an MLSDoc is permanently linked within a data structure at the security level of the edit session that created it, resulting in strongly associated security labels at byte-level granularity.

Conceived with hierarchically arranged national security classifications in mind, the MLSDoc file format is ideally suited to storing application layer files that contain nested collections of objects in a tree structure, like XML files. Figure 3 depicts a conceptual example of such a file.

In general, we require that the root object be at the lowest security level of the document, as it is required to place other objects in context. Higher-level objects will exist near the leaves of the tree, for example as annotations or additional paragraphs, etc. At the byte level of the underlying canonical representation these higher-level objects should appear to be embedded within low-level neighbours.

Figure 4 shows the MLSDoc file structure for the conceptual example in Figure 3. Data from each security level is segregated into different *Sections*, with an *Object Table* containing the information required to assemble the various pieces from each section into the application file format. Each row of the section table contains a pointer to the start of data associated with that classification, the length of that data and a version field. The version field, along with the UUID contained in the header, is used to ensure that patches are applied to the correct version of an underlying MLSDoc. The rows of the object table indicate the sequence of objects, their classifications and lengths; the pointers shown in Figure 4 are implicit, as any object follows straight after the previous one within any single security level.

Using a section based structure allows layered encryption to be applied, safely and easily embedding higher-level data within a single file. It also assists with storing the information associated with any particular security level in separate memory pages or disk blocks if desired.

IV. Trusted Processing

We required the code on the TCB that maintains separation to be minimal to allow assurance to a high level. Hence the trusted patching process does not have any application file specific awareness. The TCB simply treats each MLSDoc file as a sequence of objects, each represented as a byte string, where each object belongs to a single security level.

As indicated in Figure 2, several distinct components work in concert to provide the trusted functionality within PRISM.

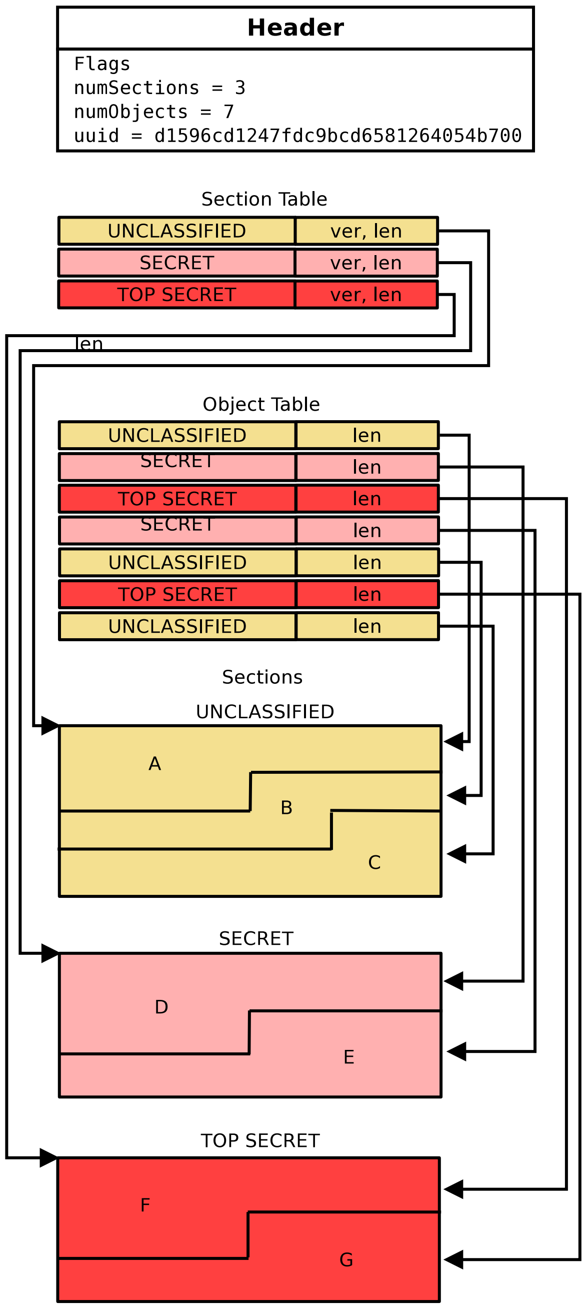


Figure 4. Actual file structure

The Verifier checks whether BLP security policy will be preserved by a patch received from an SLS partition. If so, the Patcher applies that patch to the canonical representation of a file. This file may be stored within a dedicated MLS file system or hosted using encryption in an SLS enclave. The Releaser strips high-level content to produce sanitised versions of a canonical MLSDoc that are appropriate for any target security level. This life cycle is easiest to explain starting with a trusted release.

Because the underlying MLSDoc file format separates the classification structure of the document from the data itself, trusted release to any particular security level is simple: the TCB simply strips the sections and object table entries of any higher-level objects. This mode, which we call *paranoid mode*, permits coordinated MLS file editing with zero bit-rate downwards covert channels. It provides at-level editors with read-only visibility of lower-level content and keeps them oblivious to higher-level content.

Although very usable, paranoid mode does leave the user open to making ordering errors with higher-level content. Instead, where security requirements are less stringent, the trusted release mechanism can export files in *convenience mode*, where higher-level content is instead replaced by in-line markers indicating the presence of higher-level content. This helps prompt the user to avoid damaging – or assists them to mend – any higher-level structures altered by lower-level edits. While these markers can introduce potential covert channels or may draw unwanted attention to otherwise innocuous statements, Section IX shows that the former can be managed, while protective briefings could be useful where the latter is of concern.

Lastly, the trusted releaser can export MLSDoc files in what we call *system-low* mode. Here a signed and encrypted copy of the entire canonical MLSDoc is coupled with a paranoid mode file, potentially by embedding it in an application-file specific comment field. This allows the document to be hosted within the enclave, transported to and imported through another TCB, or sent for off-line processing by legacy systems elsewhere within the enclave. For example, a system-low file may be emailed or even transferred via CD or memory stick to someone else for editing on a system not equipped with PRISM capability. When the MLSDoc returns to a PRISM capable environment any changes can be merged via the TCB into the embedded system-low version of the canonical file.

In order to explain the trusted verification and patching components it is necessary to first understand the structure of the MLS-aware patches these components are designed to receive from the untrusted diffing modules hosted in the SLS partitions.

We have adopted BSDiff [15] as a base for our PRISM patch format, since it matched our needs very well. BSDiff provides a simple representation for encoding XOR based differences (and hence *copies* in the degenerative case) and *insertions* to binary executable files. To further reduce the size of the trusted codebase, however, we constrained the existing specification to disallow differences and permit only copies and insertions.

Figure 5 shows the binary form including human readable transliteration of the BSDiff formatted patch corresponding to the delta between the Unclassified and Secret documents depicted in Figure 6. The patch consists of a header section identifying the MLSDoc and section version that the patch

```
  0: 4d 4c 53 44 49 46 46 00 61 a0 61 84 df 28 c2 86  MLSDIFF.a.a..(..
 10: 30 c3 8a 9b 01 16 48 1a 05 00 00 00 18 00 00 00  0.....H.........
 20: 00 00 00 00 20 10 00 00 6c 0d 00 00 05 01 00 00  .... ...l.......
 30: 00 00 00 00 af 01 00 00 00 00 00 00 00 00 00 00  ................

MLS Patch file
Header:
  magic:   MLSDIFF
  flags:   0x0
  uuid:    61a06184df28c28630c38a9b0116481a
  version: 5
  ctrl:  24 (0x18)
  diff:  0 (0x0)
  file:  4128 (0x1020)

Control Table:
        Diff      Extra        Skip
  -------------------------------
        3436        261           0
         431          0           0

Extra:
 40: 3c 77 3a 70 3e 3c 77 3a 70 50 72 3e 3c 77 3a 73  <w:p><w:pPr><w:s
 50: 70 61 63 69 6e 67 20 77 3a 61 66 74 65 72 3d 22  pacing w:after="
 60: 31 30 30 22 20 77 3a 62 65 66 6f 72 65 3d 22 31  100" w:before="1
 70: 30 30 22 2f 3e 3c 2f 77 3a 70 50 72 3e 3c 77 3a  00"/></w:pPr><w:
 80: 72 3e 3c 77 3a 74 3e 28 53 29 20 4f 50 45 52 41  r><w:t>(S) OPERA
 90: 54 49 4f 4e 20 46 4f 52 54 49 54 55 44 45 20 53  TION FORTITUDE S
 a0: 4f 55 54 48 3a 20 54 68 65 20 46 69 72 73 74 20  OUTH: The First 
 b0: 55 53 20 41 72 6d 79 20 47 72 6f 75 70 2c 20 63  US Army Group, c
 c0: 6f 6d 6d 61 6e 64 65 64 20 62 79 20 4c 74 2e 20  ommanded by Lt. 
 d0: 47 65 6e 2e 20 50 61 74 74 6f 6e 20 61 6e 64 20  Gen. Patton and 
 e0: 77 68 69 63 68 20 69 73 20 61 6d 61 73 73 65 64  which is amassed
 f0: 20 69 6e 20 4b 65 6e 74 2c 20 77 69 6c 6c 20 63   in Kent, will c
100: 72 6f 73 73 20 74 68 65 20 45 6e 67 6c 69 73 68  ross the English
110: 20 43 68 61 6e 6e 65 6c 20 61 6e 64 20 6c 61 6e   Channel and lan
120: 64 20 61 74 20 50 61 73 20 64 65 20 43 61 6c 61  d at Pas de Cala
130: 69 73 2e 3c 2f 77 3a 74 3e 3c 2f 77 3a 72 3e 3c  is.</w:t></w:r><
140: 2f 77 3a 70 3e                                   /w:p>
```

Figure 5. Example patch file based on difference between Secret and Top Secret views in Figure 6

should be applied to, a control table specifying the list of copies (as "diff"s with null bytes), insertions ("extra") and deletions ("skip"), and an extra section containing the strings of data to be inserted.

In this case the patch translates as: (1) copy the first 3436 bytes from the original file; (2) insert the first 261 bytes from the extra section; (3) delete the next 0 bytes that were in the original; and (4) copy the remaining 431 bytes; with no further insertions or deletions.

When a patch is received by the TCB, the trusted verifier checks that any changes to the canonical document will modify data *only* at the security level associated with that partition, thus enforcing a BLP security policy with "strong tranquillity" [5]. The content of lower or higher-level objects may not be modified. At the canonical MLSDoc level, however, lower-level objects may be split to make way for insertions of higher-level objects, and higher-level objects may be reordered as described below. Importantly, though, when using paranoid mode none of the splits or reorderings caused by an edit will be observable in the sanitised lower-level versions of the file produced by the trusted releaser; the lower-level views of a file will remain completely unchanged and hence no covert data channels are introduced.

Conceptually, the verifier will allow any sequence of objects (or parts of objects at the end points) at the security level of the current edit session to be copied or moved intact and embedded within an at-level or lower-level object anywhere else in the canonical MLSDoc. Any higher-level objects not actually present in the content of the diff transaction will be included by the TCB when applying the patch; hence higher-level objects will automatically *follow* the lower-level object in which they are embedded if the lower-level object is moved.

In paranoid mode, if the low-level bytes immediately adjacent to both sides of a higher-level object are deleted, then the higher-level object will be orphaned, as it is assumed to be highly likely that the the lower-level object in which the higher-level object resided has been deleted. These orphans could be collected at the end of each security level's section, with the COTS add-in making them available for review and recovery under interactive user control. The use of convenience mode can avoid this problem, since the TCB can be made explicitly aware of moved or deleted markers.

The patcher expands the original MLSDoc into a byte array with two interleaved vectors, one containing the data channel and the other containing the associated security labels. Next it applies the MLSDiff to the data channel, striding over the associated classification labels. Any data *copied* into the updated MLSDoc remains bytewise labelled by its original classification stream, while *inserted* data is labelled by the security level of the incoming MLSDiff.

Despite the apparent complexity of this processing, it is actually quite easy for the trusted verifier to test whether BLP strong tranquility holds for any given edit. The BLP test simply compares the before and after copies of the MLSDoc to ensure that the data content in all lower-level sections is identical; and that the order and length of object table references for lower-level objects remains unchanged. This test is required to prevent the high level from sending an arbitrary message to low by copying low-level data bytes, and ensures any reordering or deletion of low-level data is initiated from the low level.

V. Cross Domain Infrastructure

To provide a complete picture of the end-to-end processing that PRISM requires, we briefly describe the cross domain infrastructure that we used to prototype our solution on an Annex Minisec platform; although other cross domain transfer solutions such as those listed on the UCDMO Cross Domain Baseline [1] could instead be used.

The Minisec based cross domain infrastructure is constructed from a chain of components with one end starting in an untrusted partition and the other terminating in the TCB. The untrusted modules include an application-specific add-in or a file system monitor plus an application-specific translator, a user-space file system module called `mlsfs` and a communication module called `xferd`. The TCB hosts a small collection of Annex OCRM objects to marshal data and perform the requisite trusted processing. This includes a low-level communication end-point called `ChannelManager`, a partner `xferd`, a per-security level

`EnclaveHAL` that holds the authorities associated with a particular untrusted partition, cryptographic support functions, canonical file storage, and PRISM's trusted verification, patch and release modules.

The application-specific add-in fulfills two functions. It translates incoming at-level canonical MLSDoc files into an application readable format; and translates at-level edits into outgoing MLSDiff patches. Both the incoming MLSDoc files and outgoing MLSDiff patches are wrapped in simple, well-formatted `cpio` archive files that are respectively read from and saved to the `mlsfs` file system. The paired `xferd` processes ferry these transactions, in whichever direction is appropriate, between the untrusted partition and the TCB. Using `mlsfs` to decouple the application-specific add-in from `xferd` allows the add-ins to operate on top of a filesystem based abstraction. This allows the untrusted cross domain transfer code to be centralised in one place; and to also be hosted on a server within an enclave, with the untrusted add-ins interfacing to the cross domain transfer mechanism from across the network via standard file system sharing technologies like SMB or NFS.

VI. MS Word Support

Using our PRISM architecture we were able to construct an MLS document editing system using Microsoft Word as the untrusted document editor hosted in the independent system-high domains. This enables users to create fine grained MLS documents using a highly familiar COTS document editing system.

Microsoft Office 2003 introduced a single file XML format for documents called WordprocessingML. We chose this format as a basis for our work as it is human readable, and because its use of XML made it both easy to interface with MLSDoc and adaptable to future revisions of Microsoft Office or similar software.

WordprocessingML files consist of a root element `w:wordDocument` containing several schemata, whose namespaces are referred to by short prefixes on element names. The most common prefixes are `w:` (Word) and `o:` (Office). The root element and its first-level sub-elements need to appear in the document at all security levels, so these should be located at the lowest security-level of the document. Common first-level sub-elements include:

`o:DocumentProperties`
: Contains document metadata common to Microsoft Office applications such as author, title, word count and edit time.

`w:fonts`
: Contains the set of fonts used in the document.

`w:styles`
: Contains the set of styles used in the document.

`w:docPr`
: Contains document property metadata specific to Microsoft Word.

`w:body`
: This is where the main content of the document appears and is therefore the part that most needs to contain data at differing security levels. It is the only essential component of the document.

The top-level metadata objects contained within the root element maintain settings for features used in the rest of the document. If these are treated as MLS there is the possibility that a feature introduced at a high level first (for example tables) but later used at the low level may result in ambiguous duplication that might confuse Word's operation.

A simple, low-tech solution to this problem is to simply create documents at the lowest level using a template that contains any features that will be used.

A. MLSDiff Patch generation

To generate an MLSDiff patch for an updated WordprocessingML file, the untrusted add-in transforms the new XML document into a canonical format and compares it with the old one from the MLSDoc container that it originally received from the TCB. Rather than using the BSDiff algorithm to generate the patch directly, the untrusted add-in uses an XML aware diff algorithm to detect changes and then translate these into the simple MLSDiff patch format for further processing by the TCB.

We adapted a C♯ .NET XML difference engine called XML Diff [16] for this purpose. This implements an XML tree edit distance algorithm that generates an XML Diff Language (XDL) diffgram of the differences between the two versions of an XML document. We selected this package because it is able to represent subtree moves, compares well with other algorithms in terms of performance and precision [19], and because its .NET implementation is well suited to incorporation in a Word add-in.

B. File Format Work-Arounds

Because of the cross-linked structure of WordprocessingML files, we did encounter several cases where user edits initiated at a higher security level would modify formatting instructions stored at lower security levels of the document. These modifications would then become embedded in the next MLSDiff patch sent to the TCB, causing a BLP policy violation and patch rejection.

We found that these problems could be worked around by simply filtering out, or otherwise modifying, problematic XML tags and allowing Word to reconstruct these elements at file load time. This was accomplished using only 100 lines of XSLT. Listing 1 shows a fragment of this XSLT stylesheet containing representative templates for achieving these transformations. While the modifications we discuss consist of only a selection of the issues triggered by the test documents and version of Word that we used in development, we believe they are indicative of the types of issues

```
<!-- Remove unsupported tags -->
<xsl:template match="/w:wordDocument/
o:DocumentProperties/o:LastSaved"/>

<!-- Sort w:wordDocument children into a
      consistent order. In particular sort
      o:SmartTagType by name attribute. -->
<xsl:template match="w:wordDocument">
  <xsl:copy>
    <xsl:for-each select="@*">
      <xsl:sort select="name()"/>
      <xsl:copy/>
    </xsl:for-each>

    <xsl:for-each select="o:SmartTagType">
      <xsl:sort select="@o:name"/>
      <xsl:copy-of select="."/>
    </xsl:for-each>

    <xsl:for-each select="*">
      <xsl:if
          test="name() != 'o:SmartTagType'">
        <xsl:copy>
          <xsl:apply-templates/>
        </xsl:copy>
      </xsl:if>
    </xsl:for-each>
  </xsl:copy>
</xsl:template>

<!-- Default template just copies everything
      intact and sorts the attributes -->
<xsl:template match="*">
  <xsl:copy>
    <xsl:for-each select="@*">
      <xsl:sort select="name()"/>
      <xsl:copy/>
    </xsl:for-each>
    <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>
```

Listing 1. XSLT Stylesheet fragment for transforming a Microsoft Word 2003 WordprocessingML formatted document into an MLSDoc compatible form.

that would need to be dealt with to fully adapt Word for MLSDoc compatibility.

Problem: Node `/w:wordDocument/o:Document-Properties` has the following fields that are frequently updated:

`o:Created`
: When the user selects "save as".

`o:Revision`
: Incremented each time the document is saved.

`o:LastSaved`
: Updated each time the document is saved.

`o:LastPrinted`
: Updated with the last time the document was printed if it was printed during an edit session.

`o:TotalTime`
: Maintains the total editing time.

`o:Characters`
: When the number of non-space characters in the document changes.

`o:CharactersWithSpaces`
: When the number of characters including spaces in the document changes.

`o:Words`
: When the number of words in the document changes.

`o:Paragraphs`
: When the number of paragraphs changes.

`o:Lines`
: When the number of lines changes.

`o:Pages`
: When the number of pages changes.

Solution: The `o:Created` tag should be created at the lowest security level of the document. The `o:LastSaved`, `o:LastPrinted` and `o:TotalTime` tags must be deleted, or copied from the old version in all but the lowest security-level copy of the document. The `o:Revision` tag must be deleted prior to storage inthe MLSDoc container, but the tag can be generated prior to loading the MLSDoc into Word by substituting the revision numbers for each security level in the MLSDoc file.

The other statistical properties tags are recalculated when Word loads the document and may be safely removed for storage in the MLSDoc container.

Problem: When editing text at a higher level, any new styles or fonts that are added to the document are added as `/w:wordDocument/-w:lists/w:listDef/w:lvl/w:pStyle` and `/w:wordDocument/w:fonts/w:font`. If the same document is edited at a lower level and a new style added, there is the possibility of a style name being used more than once, possibly confusing Word and/or the user.

Solution: Add all styles at the lowest security level of the document to ensure that they are available at all security levels and avoid name clashes, for example using a template as already discussed.

Problem: When Word loads and saves an XML document, it sometimes alters the order of tags for some XML entities. For example, as shown in Table I, we observed a document containing a number of `o:SmartTagType` entities as the first children of the `w:wordDocument` entity being reordered when loading and immediately saving the document using the "save as" option.

Solution: Define a canonical order for these elements, and to transform the document into this order. XSLT code to do this using alphanumerical ordering of SmartTag elements according to their *name* attributes is included in Listing 1.

Table I
REORDERING OF `SmartTagType` TAGS DURING AN EDIT SESSION

`o:name` attribute	original order	saved order
PlaceName	1	2
PlaceType	2	3
Street	3	7
City	4	4
State	5	1
country-region	6	6
place	7	5
address	8	8

Problem: New metadata entity tags appear and disappear from time to time when editing documents.

Solution: Optional tags, such as `w:rsid` and `w:proofState`, should be deleted prior to MLSDiff generation for storage in an MLSDoc container.

Problem: The `aml:annotation` element represents a tracked insertion, deletion, formatting change, comment, or bookmark in a document. It contains the attribute `aml:id` which is numbered sequentially in the document. Any insertion or deletion of an `aml:annotation` element will alter the numbering of all subsequent `aml:annotation` elements, causing BLP policy violations and TCB patch rejection.

Solution: The `aml:annotation` element's `aml:id` attribute can be stripped from these tags prior to insertion into or diff against the MLSDoc file. When the file is subsequently loaded and saved by Word, some of these elements may be lost. For example, insertions have a `w:type` attribute of `"Word.Insertion"` and are retained by Word when the document is loaded and saved again. A new `aml:id` attribute is generated if it is missing from the source document. Bookmark tags with a `w:type` attribute of `"Word.Bookmark.Start"` or `"Word.Bookmark.End"`, however, are not subsequently saved by Word. These elements need to be removed from the document prior to storage in an MLSDoc container.

Problem: Third party applications such as OpenOffice, and even different versions of Word, typically write files using different detailed representations of the information in a file. For example, the order of entity and attribute tags is not typically maintained, numerical values may be formatted differently, and even XML structures may vary. Listing 2 shows three common ways that we have encountered for representing empty XML tags.

```
<entity></entity>
<entity/>
<entity />
```

Listing 2. Three common representations for empty XML tags

While these alternate representations are still semantically valid and should be understandable by Microsoft Office and other applications, the syntactic differences foil MLSDoc's strict byte based BLP policy checks.

Solution 1: If transitioning to new application software at all security levels, a one-off reconstruction of the document can be achieved by transferring to the new application at the lowest security level and migrating upwards through each level, manually importing higher-level content.

Solution 2: An alternate approach would be to have a much tighter canonical form that the document is transformed to. This could be achieved by improving file format standards, or with more extensive XML pre-processing.

Problem: High-level cross references to low-level objects will break if the low-level objects are renamed or removed, which would be particularly problematic if low-level labels were reused.

Solution 1: Similar to the discussion about orphaned fragments in Section IV, the untrusted add-in could draw any broken references to the user's attention and assist them to repair any broken associations.

Solution 2: A better solution would be for the low-level add-in to encode remapping information into the document for transmission to the high-level add-in which would enact any necessary substitutions. This could perhaps be achieved by an intelligent low-level differ, but a more powerful option is canvassed in Section X on Future Work.

C. PRISM Add-in

We have developed a .NET based PRISM add-in for Word using Microsoft VisualStudio Tools for Office (VSTO) to automate MLSDoc processing within a standard "creation/load; edit; save" workflow. While the use of Microsoft's COM XSLT processor appeared to be a natural fit for our needs, in practise it interfered with XML formatting so we instead used XMLStarlet for our XSLT processor.

The PRISM Word add-in provides a "create new document" dialog box that allows the user to create a new MLSDoc. It also provides an "open existing document" dialog box that allows the user to open an existing MLSDoc container. In this case, the MLSDoc is assembled into a Word 2003 WordprocessingML file and loaded. The add-in then makes use of Word's standard document protection mechanisms to insert read-only tags throughout the document to prevent accidental alteration of lower-level data, which would be subsequently rejected by the TCB.

A "save document as" dialog box is also provided, which intercepts document save operations to first carry out the following transformations: deletion of any problematic XML entities; re-ordering attributes into a canonical alphanumeric ordering; XDL diffgram generation by comparison with the original assembled document; and MLSDiff patch generation from this XDL diffgram.

Following this the `xferd` process described in Section V sends the MLSDiff patch to the TCB for verification, patching and redistribution. The Word add-in meanwhile monitors the file system so that it can reload the updated MLSDoc when it is received from the TCB in response to this, or changes from another security level.

VII. A Seamless User Experience

By judiciously restricting how the user can interact with the system it is possible to make multiple SLS instantiations of a COTS application appear to behave as a single MLS instantiation, thus delivering a seamless user experience almost indistinguishable from native MLS. Although we demonstrate this in the context of using Microsoft Word to edit an MLSDoc on an Annex Minisec platform, our method is also suitable for integrating other COTS applications that have been replicated inside a separation kernel or virtualisation based isolation environment.

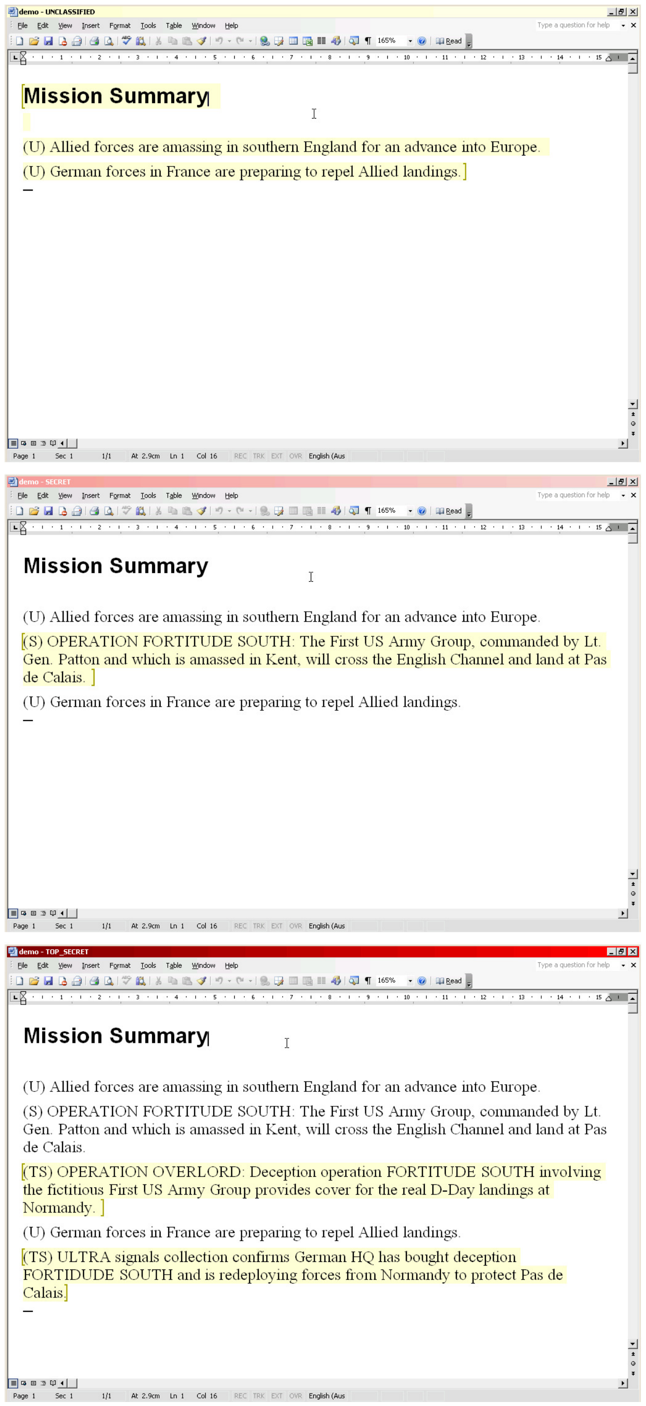
As introduced in Section II, the Minisec's TCB mediates each MILS partition's access to a display, mouse and keyboard. In its current version, Minisec3, the TCB instantiates three minimal VNC [17] clients in separate Annex OCRM objects, each connecting through the Minisec's labelled networking mechanism to a VNC server running in a different hardware isolated MILS partition. Each VNC client interacts with a separate, dedicated TCB-virtualised frame-buffer, mouse and keyboard, only one set of which is synced through to the real underlying hardware at any point in time, depending on the state of the Minisec's trusted buttons. These LED-lit push-button switches, one per MILS partition, provide a simple and secure means for the user to unambiguously identify and/or alter the active partition with which they wish to interact.

Coupled with our PRISM architecture, the Minisec's secure KVM-like functionality provides an excellent basis for providing a seamless, MLS-like user experience. Figure 6 demonstrates this seamlessness in the context of editing a Microsoft Word document in a national security classified environment; the three separate sub-figures show the view of a single document being edited at Unclassified, Secret and Top Secret respectively.

A. *Automatic sync-and-switch*

We have extended a Minisec user's ability to switch between partitions using a trusted button by first performing a synchronisation step when they "double-press". This causes the the TCB to send a message initiating a document-save in the original partition; switch the keyboard, mouse and video as normal; and – when in convenience mode or when changing to a higher level in paranoid mode – send another message causing the document to be reloaded in the newly active partition. Ideally the cursor would be repositioned so that the user can continue editing either at or near where they left off, although we have not yet implemented this additional functionality.

Furthermore, by carefully pre-arranging a consistent window geometry between partitions – for example by using


demo - UNCLASSIFIED
File Edit View Insert Format Tools Table Window Help
Type a question for help
165%
Read
Page 1 Sec 1 1/1 At 2.9cm Ln 1 Col 16 REC TRK EXT OVR English (Aus
demo - SECRET
demo - TOP_SECRET

Mission Summary

(U) Allied forces are amassing in southern England for an advance into Europe.

(U) German forces in France are preparing to repel Allied landings.

Mission Summary

(U) Allied forces are amassing in southern England for an advance into Europe.

(S) OPERATION FORTITUDE SOUTH: The First US Army Group, commanded by Lt. Gen. Patton and which is amassed in Kent, will cross the English Channel and land at Pas de Calais.

(U) German forces in France are preparing to repel Allied landings.

Mission Summary

(U) Allied forces are amassing in southern England for an advance into Europe.

(S) OPERATION FORTITUDE SOUTH: The First US Army Group, commanded by Lt. Gen. Patton and which is amassed in Kent, will cross the English Channel and land at Pas de Calais.

(TS) OPERATION OVERLORD: Deception operation FORTITUDE SOUTH involving the fictitious First US Army Group provides cover for the real D-Day landings at Normandy.

(U) German forces in France are preparing to repel Allied landings.

(TS) ULTRA signals collection confirms German HQ has bought deception FORTIDUDE SOUTH and is redeploying forces from Normandy to protect Pas de Calais.

Figure 6. Unclassified, Secret and Top Secret views of an MLS Word document being edited in paranoid mode.

full-screen operation – switching security levels appears to simply refresh which data is currently visible to the user. All other user interface elements remain essentially intact and consistent. In addition, though, our untrusted PRISM add-in also applies thematic window colourisations (in sync with the TCB lighting the appropriate trusted button) to help the user quickly identify the new security level at which they are now operating.

Our untrusted add-in also makes use of Word's built-in document protection methods to mark any text at the current classification read/write, while other sections are marked read-only. Word highlights the read/write sections in yellow, which clearly identifies the at-level sections to the user and helps them only edit those parts of the document that the TCB-enforced BLP security policy verification step will allow them to change.

These various indicators greatly decrease the cognitive load placed on a user when switching domains, allowing them to flit effortlessly between security levels with simply the (double) press of a button.

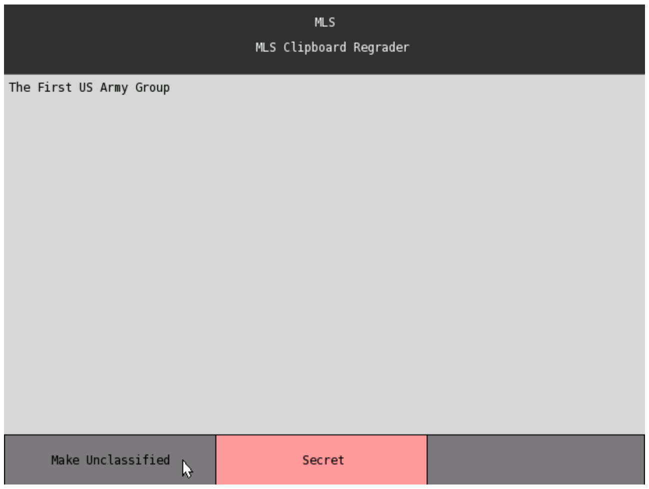
B. MLS copy-and-paste

The VNC protocol's *cuttext* message type [17] provides a simple mechanism for sharing clipboard buffer contents between an untrusted VNC server and its corresponding VNC client. We used this to provide MLS copy-and-paste between MILS partitions by developing an OCRM-protected communication channel that permits BLP information flow between each VNC instance's clipboard.

After an object is copied into a VNC server's clipboard and received by the corresponding VNC client, the TCB automatically copies the clipboard contents to all higher-level VNC clients and hence to the VNC servers in the corresponding MILS partitions. The user can then switch to any higher-level partition and paste the updated clipboard contents into any supported application. This provides a highly convenient and intuitive mechanism for regrading content and sharing it between unmodified COTS applications running at different security levels.

We have also developed an OCRM based downgrade application, shown in Figure 7, that allows a user to view text based clipboard contents in a trusted, TCB-hosted viewer and manually regrade the content to a lower level.

This figure shows a user downgrading a fragment of secret text (from Figure 6) to unclassified, so that it may be pasted from the MLS clipboard into an unclassified document.



The First US Army Group

Figure 7. A trusted application facillitating user review and downgrade of textual copy-and-paste information.

C. MLS Filing System

To help the users organise their MLS files, the `mlsfs` module introduced in Section V manages several namespaces to provide users with a secure and intuitive view of where files originated and what higher levels of data have been added to them.

To minimise user confusion, we aimed to provide a synchronised namespace for all files across all levels. However, in order to prevent filenames from being used as a covert channel our TCB-enforced security policy insists that file creation (or renaming) may only take place at the lowest security level at which that file will exist. Lower levels should remain oblivious of such a file's existence, and higher-level partitions may only embed further data content *within* the file. These conditions resonate well with a BLP security model with strong tranquillity and our requirement that "the root object be at the lowest security level of the document" from Section III.

With these restrictions in mind, `mlsfs` polyinstantiates lower-level directory trees upwards to higher levels. For an unclassified partition this results in a single directory tree, `tree1` rooted like this:

```
/mls/unclassified/<tree1> (data:rw, namespace:rw)
```

The view from a secret-level partition contains two trees:

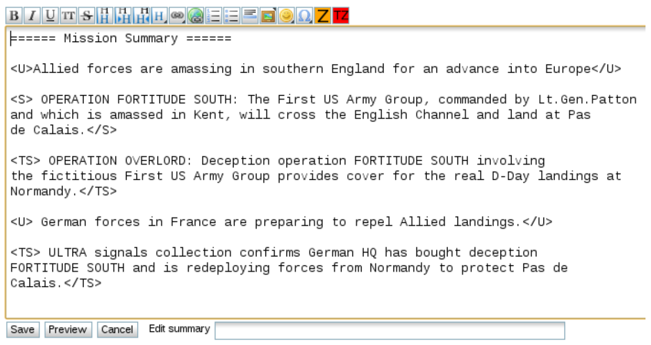
```
/mls/unclassified..secret/<tree1> (data:rw, namespace:ro)
/mls/secret/<tree2> (data:rw, namespace:rw)
```

where `tree1` is a polyinstantiation of the unclassified tree. The root path `unclassified..secret` indicates that the *namespace* is maintained by the `unclassified` partition (and is hence read-only at this level), but that the files may contain additional *content* ranging up to `secret`. This arrangement permits MLS documents containing unclassified information to include secret-level content. The second tree, `tree2` provides a mechanism for creating files containing a minimum of secret-level data, and whose very existence is even kept secret from lower levels.

The view from a top secret partition is similarly constructed:

```
/mls/unclassified..topsecret/<tree1> (data:rw, ns:ro)
/mls/secret..topsecret/<tree2> (data:rw, ns:ro)
/mls/topsecret/<tree3> (data:rw, ns:rw)
```

This enumeration of multiple hierarchical namespaces that are rooted by an originating classification provides a secure, clean and flexible abstraction on which larger scale MLS document management can be easily managed by users or higher-level PRISM-aware software.



```
====== Mission Summary ======

<U>Allied forces are amassing in southern England for an advance into Europe</U>

<S> OPERATION FORTITUDE SOUTH: The First US Army Group, commanded by Lt.Gen.Patton
and which is amassed in Kent, will cross the English Channel and land at Pas
de Calais.</S>

<TS> OPERATION OVERLORD: Deception operation FORTITUDE SOUTH involving
the fictitious First US Army Group provides cover for the real D-Day landings at
Normandy.</TS>

<U> German forces in France are preparing to repel Allied landings.</U>

<TS> ULTRA signals collection confirms German HQ has bought deception
FORTITUDE SOUTH and is redeploying forces from Normandy to protect Pas de
Calais.</TS>
```

Figure 8. The Top Secret view when editing an MLS DokuWiki.

VIII. DokuWiki Support

We have used PRISM's MLS filing system to store fined-grained MLS information within DokuWiki [8]. As with all applications that could be built on top of PRISM, the trusted processing remains the same; only a new untrusted diffing component needed to be developed. However, our careful choice of DokuWiki – which uses plain text XML compatible files for wiki page storage – allowed us to reuse much of the same untrusted diffing engine that we used for the Microsoft Word add-in.

Using the `mlsfs` system to intercept application file changes allowed us to avoid making any invasive changes to the DokuWiki source code, demonstrating a greater degree of decoupling between MLS infrastructure and the native application than was the case with MS Word. When a modified wiki file is saved, the `mlsfs` system uses XMLStarlet to convert any changes into MLSDiff patches and forwards them on to the TCB for trusted BLP policy verification, merging into the file's canonical MLSDoc container and the sanitised released back to all partitions.

On arrival in a partition the `mlsfs` system notifies the DokuWiki plugin, which assembles the changes back into DokuWiki native file format.

Figure 8 demonstrates a wiki version of the Top Secret view of the MLS Word document shown in Figure 6.

IX. Security Properties

This section provides a critical analysis of the security properties of our PRISM architecture. Since data *confidentiality* was our primary concern, we have paid particular attention to identifying and mitigating potential Covert Channels (CC). Secondary concern for data *integrity* and *availability* encouraged us to examine the potential susceptibility of our system to Replay Attacks and classification spoofing that might confuse the user. Finally, while somewhat beyond the scope of this paper, we briefly comment on how implementing PRISM on top of the Minisec's Silicon Trojan resistant design provides further guarantees for confidentiality, integrity and availability.

We finalise our discussion on security properties by arguing the high tractability of certifying products based on our architecture.

A. Covert Channels

The Trusted Computer Security Evaluation Criteria (TCSEC) [7] define two types of covert channel: storage channels and timing channels. Both aim to hide the transfer of information between two entities, in either an unintended side channel or encoded within an otherwise legitimate data stream. Storage channels hide information by directly modifying typically unused or redundant elements within a data stream, while timing channels signal information through the relative timing or ordering of events.

In this discussion we ignore any covert channels in the underlying MILS or Annex separation kernel, since a primary function of this kernel is to provide strong protection against these, and focus only on additional channels relating to the timing and content of file updates introduced by distribution of the MLSDoc files.

PRISM's susceptibility to such covert channels differs depending on whether it is used in paranoid mode or convenience mode, plus whether or not system-low storage is used. In all cases, however, PRISM's use of opaque data streams restricts covert channels to modulating *metadata* such as when files are synchronised, their path, their size or the location of convenience mode markers. This drastically reduces PRISM's susceptibility to high capacity covert channels.

1) Paranoid Mode: Assuming the proper operation of the TCB, PRISM's paranoid mode without system-low storage results in no high to low information flows of any kind; hence there can be no covert channels.

2) Common to System-Low and Convenience Modes: Both system-low and convenience modes involve updating lower-level copies of an MLSDoc file.

A potential covert channel would exist if untrusted software were allowed to control the timing of synchronisations, thus creating a timing channel.

The potential capacity of such a channel can be approximated by modelling it as a discrete noiseless channel driven by a Markov process where the entropy of the discrete random variable X is given by $H(X) = -\sum_{x \in X} p(x) \log p(x)$ [20]. A binary random variable then models the presence, or otherwise, of a synchronisation occurring during a given time slot of length τ seconds. If the rate of synchronisations is limited to R per second where $R\tau \ll 1$ per slot then the probability of a synchronisation occurring within a given time slot $p(x = \text{sync}) \approx R\tau$, and conversely $p(x = !\text{sync}) \approx (1 - R\tau)$. Hence, the potential timing covert channel capacity of $\frac{1}{\tau}H(X)$ bits per second is then:

$$CC_{timing} \approx -\frac{1}{\tau}\left(R\tau \log_2(R\tau) + (1 - R\tau)\log_2(1 - R\tau)\right)$$

If the TCB enforced a timing resolution of 1 second and a limit of 100 synchronisations per day (1.16×10^{-3} per second), a covert timing channel could potentially transmit up to 140 bytes per day.

However, rather than having the TCB blindly redistribute high-level updates to lower levels, the sync-and-switch protocol from Section VII limits updates to whenever the user wishes to synchronise content and switch to a different domain. This prevents untrusted software from modulating synchronisation operations and the timing channel is completely removed.

Further covert channels would exist if the untrusted software were allowed to synchronise any filename in any order, using the filename or ordering to contain or modulate a hidden message. These can also be completely countered, however, by TCB GUI software indicating the file being synchronised and asking the user to confirm this action; an example of such a system will follow shortly. The attempted operation of a such a covert channel would become rapidly apparent to the user if the system started trying to synchronise files that were not being actively edited.

3) Unique to System-Low Mode: The use of system-low storage potentially allows a high-level process to encode a message readable at the low level by modulating the length EL bytes of an MLSDoc's encrypted content. However, the potential channel capacity of this scheme can be extremely constrained by padding the encrypted section's length to a power of two, reducing the channel capacity to $\log_2(\log_2(EL))$ bits per synchronisation. For example, for files "limited" to between a 32 byte (2^5) cipher block and 2^{261} bytes of encrypted content (i.e. practically unlimited) this results in a potential covert channel of one byte per synchronisation. As above, trusted sync-and-switch can limit the number of synchronisations, and TCB GUI software can help flag wild file size variations to the user.

4) Unique to Convenience Mode: Convenience mode introduces the most problematic potential covert channel. An inherent side effect of the desirable functionality that convenience mode affords is that there is also scope for malicious untrusted software to modulate a covert channel using high-level content markers between each byte of lower-level data. This simple scheme could yield the transmission of one bit of information per byte of low-level data synchronised.

This dangerously high capacity channel can be easily constrained, however, by allowing for only a reasonable number M of high-level marker insertions or deletions per synchronisation. The potential capacity will then be bounded by the entropy present in the variation of where high-level markers can potentially be inserted in (or removed from) low-level data. Given L bytes at the low security level there are $L + 1$ locations where markers may be inserted (or removed). Modifying m markers during any individual synchronisation results in C_m^{L+1} possibilities. The total number of symbols that can be encoded is then the sum

Table II
UPPER BOUNDS IN BYTES PER SYNCHRONISATION OF POTENTIAL COVERT CHANNEL WHEN IN CONVENIENCE MODE AND ALLOWING THE MODIFICATION OF UP TO M MARKERS WITHIN L LOW-LEVEL BYTES USING EQUATION 1.

L	Markers, M						
(bytes)	1	5	10	50	100	200	1000
100	0.83	3.29	5.54	12.5	12.6	12.6	12.6
1000	1.25	5.37	9.73	35.3	58.1	89.7	125
3873	1.49	6.59	12.2	47.7	83.2	141	398
10000	1.66	7.44	13.9	56.3	100	176	586
100000	2.08	9.52	18.0	77.0	142	260	1009
10^6	2.49	11.6	22.2	97.8	184	343	1425
8×10^6	2.87	13.5	25.9	117	221	418	1800
10^7	2.91	13.7	26.3	119	225	426	1841
10^8	3.32	15.7	30.5	139	267	509	2256
10^9	3.74	17.8	34.6	160	308	592	2671

of possible marker combinations for all $m \leq M$, leading to a per-synchronisation potential covert channel (in bits) of:

$$CC_{convenience} = \log_2 \left(\sum_{m=0}^{M} \frac{(L+1)!}{(L+1-m)!m!} \right) \quad (1)$$

This is easily computable in O(M) operations by starting at $m = 0$, noting that $C_{m+1}^{L+1} = \frac{L+1-m}{m+1} C_m^{L+1}$ and rescaling if necessary to avoid overflow.

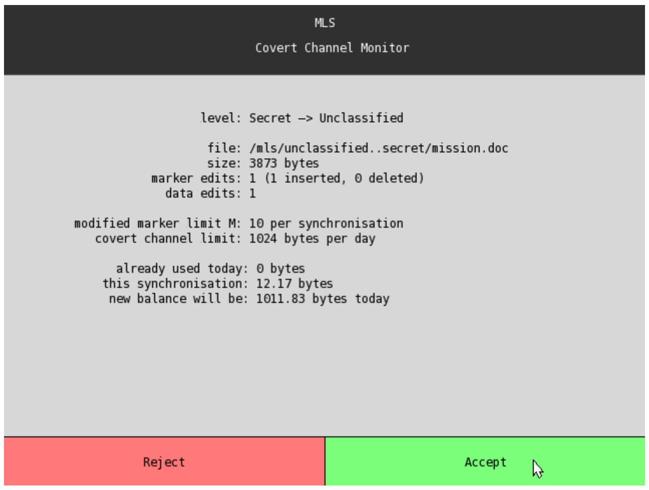
Alternatively, the covert channel can be estimated as per the timing channel by modelling it as a Markov process filling $L + M$ slots with either a data byte (X=0) or marker (X=1). The channel capacity in bits per synchronisation is bounded by the entropy H as follows:

$$CC_{convenience} \leq L \log_2 \frac{L+M}{L} + M \log_2 \frac{L+M}{M}$$

This is an overestimate because it allows for files containing adjacent markers and for files with greater than M markers, both of which are precluded by the TCB.

With trivial differences, applying either of these final equations can give a theoretical upper limit for the potential covert channel capacity of convenience mode. Upper bounds for various combinations of M and L are given in Table II.

While these are theoretical limits, even simple signalling schemes can achieve high efficiencies when $M \ll L$, especially when the preservation of legitimate user data can be dispensed with – although the latter will obviously destroy any chance of stealthy operation. For example, if the low-level file were conceptually split into $\frac{L}{M}$ blocks, the careful positioning of one marker in each block could yield a total covert channel capacity of $M \times \log_2 \frac{L}{M}$ bytes per synchronisation. For an 8MB document limited by up to 10 high-level content marker modifications at a time, this scheme would yield 24.6 bytes per synchronisation, which is 95% of the theoretical peak of 26 bytes.



```
                     level: Secret -> Unclassified

                      file: /mls/unclassified..secret/mission.doc
                      size: 3873 bytes
              marker edits: 1 (1 inserted, 0 deleted)
                data edits: 1

   modified marker limit M: 10 per synchronisation
      covert channel limit: 1024 bytes per day

        already used today: 0 bytes
      this synchronisation: 12.17 bytes
       new balance will be: 1011.83 bytes today
```

Figure 9. A trusted application for displaying metadata and potential covert channel implications associated with a particular synchronisation operation, for user review and acceptance.

Since the TCB mediates all document synchronisations it can easily monitor potential covert channels to maintain them within policy based limits. Figure 9 shows a prototype covert channel monitor that we have developed to demonstrate the utility of this concept. While this uses the policy limit for the number of markers per file to calculate the potential information content it could be extended to provide a tighter bound based on the actual data observed [21].

This shows a TCB produced dialog box of what the user would see – if they were in convenience mode with $L = 3873$ and M limited to 10 – after having just entered the secret text shown in Figure 6 and then requested synchronisation. The user should verify that the metadata corresponds to the synchronisation action that they intended to make, hence accepting or rejecting the operation.

In an intelligence community environment this trusted application could enforce a covert channel policy demanding the use of 0 bits per second paranoid mode; in other military scenarios it may apply rate limiting to maintain potential covert channel bitrates below an allowable threshold; while in commercial settings where such sophisticated attacks are less likely it might simply quantify the potential leakage and present this to the user for monitoring purposes.

B. Replay Attack

A user or untrusted software may attempt to mount a *replay attack* by forwarding to the TCB MLSDiff edits based on an out of date version of an MLSDoc, attempting to overwrite documents at other levels with old information. Such an effort would be thwarted when not using system-low mode, however, since the canonical MLSDoc contains revision numbers that protect the version-integrity of each security level's section; a stale MLSDiff patch would not match the current version of the canonical document and the TCB would simply reject the changes. To counter this problem in system-low mode the TCB would need to maintain a version identifier for each system-low file, although the data remains elsewhere.

C. Classification Spoofing

A *classification spoofing* attack against PRISM involves attempting to fool a user or untrusted software about the classification of particular content by formatting it so that it appears to belong to a different security level. For example, top secret content may be formatted (in the top secret domain) to appear as though it is unclassified in the hope that the user may inappropriately treat it as such and re-divulge that information at a lower level. Alternatively, unclassified content may be formatted (in the unclassified domain) as top secret in the hope that a high-level user or untrusted software will inappropriately treat it as such and grant it more weight than it deserves.

In all of these cases, however, it is important to remember that regardless of what formatting any untrusted software in an SLS partition applies to the data entered into it, MLSDoc's trusted processing ensures that the data (and formatting instructions) remain inextricably linked to the security level at which they were created. Hence, the example misattributed "unclassified" information held at the top secret level and the "top secret" information found at the unclassified level actually retain their confidentiality and security-level integrity within the technical confines of the PRISM system. If automated software becomes confused and attempts to patch this data through MLSDoc to a different security level, the changes will be rejected. As is the case in existing system-high environments, an unconfined person should remain on guard for these types of psychological attacks; PRISM helps them identify such misinformation by simply viewing the same document at different security levels and noticing when information purporting to be of a low security level disappears when switching down to that level.

D. Silicon Trojans

There is a growing threat from Silicon Trojans that contain malicious behaviour embedded in compromised hardware [4]. This type of malware is arguably impossible to counter with software based isolation mechanisms running on COTS hardware. While PRISM does not counter this threat directly, our implementation on top of the Minisec's hardware isolated MILS domains goes some way to reducing susceptibility to compromised hardware and untrusted software alike. Even if data or processing within a SLS partition is compromised, the effects cannot leak to lower security-level enclaves other than by bounded covert channels through the TCB, and a high degree of confidentiality and security-level integrity are preserved. These in turn improve *availability* by frustrating an adversary's ability

Table III
TRUSTED CODEBASE SIZE FOR PROTOTYPE AND OPTIMISED VERSIONS; AND UNTRUSTED CODEBASE SIZE FOR THE MS WORD USE CASE.

	Component	Language	Lines of Code	Total LOC
Untrusted	PRISM add-in	C	8200	
		C♯	2750	
	XML Library	C	195000	
	XML Diff Library	C♯	9000	
	Windows, Office, .NET framework	C/C++/C♯	$>$ 10,000,000	
	xferd	C	1259	
	libfuse	C	13974	
	mlsfs	C	732	unimportant
Trusted (Prototype)	Verify/patch/release	C	1600	
	crypto	C	27038	
	EnclaveHAL	C	420	
	xferd	C	1259	
	ChannelManager	C	3375	
	Annex OCRM	C	8616	
	Minisec3 VNC clients	C	1264	
	OCRM based GUI	C	5339	
	OCRM based downgrader	C	399	
	OCRM based CC policy checker	C	151	
	Linux kernel	C	$>$ 1,000,000	kernel + 49,461
Trusted (Minimal Optimised)	Verify/patch/release	C	1600	
	certified crypto implementation	C	<2000	
	Communication	C	250	
	Green Hills Integrity or seL4	C	<10000	μkernel + crypto + 1850

to construct a robust command and control channel for triggering a *Denial of Service* (DoS) attack.

E. Evaluation and Certification

PRISM relies on only a very a small amount of code in the TCB to provide its security critical verification, patch and release functionality. Furthermore, the same trusted codebase is common for any MLSDoc based use case; the untrusted add-in modules abstract away any application-specific file format issues. This makes is both feasible and worthwhile to evaluate PRISM's trusted components to very high levels of assurance.

Indicative code sizes for various parts of our PRISM system are shown in Table III. The first group of components covers the untrusted COTS operating system, application software and custom cross domain infrastructure residing in an SLS MILS partition. From a security perspective, these untrusted elements are assumed to be capable of any possible (mis)behaviour. From a modelling perspective the size of this codebase is therefore unimportant.

The second group of components includes the code for all trusted elements in our current Minisec based prototype implementation. With the exception of the very large Linux kernel – but not a standard Linux user-space, which is replaced by the OCRM-hosted PRISM implementation – this totals just 42,308 lines of trusted code. However, although 2-3 orders of magnitude smaller than the untrusted code base, even this is beyond the current reach of exhaustive formal modelling techniques.

The third group of components represents our estimation of the minimum sized code base that could implement PRISM's core functionality in paranoid mode using only symmetric cryptography, optimising it for productisation. This disables the identified covert channels along with the requirement for monitoring them, and excludes public key cryptography required for sharing files between TCBs. The Linux kernel could be replaced with a highly certified COTS separation kernel such as Green Hills Integrity or seL4. The general purpose MLS middleware and cross domain infrastructure would be replaced by a minimal, single purpose framework built for solely supporting a likewise minimised PRISM implementation. Hence, we estimate the entire TCB could be reduced to less than 10,000 lines of (pre-certified) code for the μkernel base and crypto support, and less than 2,000 lines for PRISM application component.

A thorough security analysis of this optimised PRISM implementation would be eminently achievable using current state of the art modelling techniques. With careful implementation and evaluation, we believe that certification to EAL6+ or beyond of PRISM's extremely useful capability would be a highly tractable proposition.

X. Future work

Although we have already demonstrated the feasibility of adapting complex COTS software to a high assurance MLS environment, an improved approach would include building MLS *awareness* (but not trust) into COTS applications. For example, in order to assist the TCB to differentiate between a copy/paste that may contain embedded high-level data and an at-level insertion of new data, or to expose events such as the renaming of low-level cross reference structures, the patch should be constructed as a transaction of the actual operation being performed by the application. These types of extensions would encourage more frequent, smaller transactions, help avoid ambiguous corner cases, and lead to more robust operation. Sun *et. al.* [22] demonstrate an adaptation layer for Microsoft Office that intercepts mouse and keyboard input which could be used to implement this type of facility. While this would add complexity to the untrusted application, it is outside the TCB and would not require difficult and expensive security evaluation.

Our discussion in this paper also glossed over cryptographic mechanisms for signing, encryption and key distribution. Although the Annex OCRM provides methods for public and private key cryptography that are used for the authentication and encryption of communication between Minisec devices, we have not yet integrated the use of these primitives for key agreement, signatures or encryption within MLSDoc files. It is not difficult, however, to envisage leveraging these facilities so that one Minisec's TCB can export an MLSDoc to another TCB by block encrypting the MLSDoc with a random key and distributing that package to the target TCB along with a public-key encrypted bulk decryption key. A similar mechanism could be used to protect canonical MLSDoc files stored in system-low mode outside the TCB.

XI. Conclusion

We have shown how our Program Replication and Integration for Seamless MILS (PRISM) architecture can maintain trusted, fine grained (intra-document) Bell LaPadula separation between information at multiple security levels. The security of our solution relies on only a tiny Trusted Computing Base (TCB) to maintain the canonical representations of Multi-Level Secure (MLS) documents and provide a policy-verifying update and coordination mechanism. When implemented on the Annex Minisec platform, these mechanisms provide strong guarantees on data confidentiality, integrity and availability. All other functionality is provided by Commercial Off The Shelf (COTS) software and file formats, plus small untrusted add-ins for performing file format translation. We presented the utility of our approach using Microsoft Word and DokuWiki.

We also showed that by judiciously restricting user interaction, multiple Single-Level Secure (SLS) instantiations of COTS programs can be made to appear to behave as a single MLS instantiation, thus delivering a seamless user experience almost indistinguishable from native MLS. Although we demonstrated this using the Annex Minisec platform, our method is equally suitable for retrofitting MLS functionality into other COTS applications using a Multiple Independent Levels of Security (MILS) framework in any separation kernel or virtualisation based isolation environment.

Acknowledgments

We wish to thank the anonymous reviewers for their helpful comments that assisted in clarifying aspects of the paper.

References

[1] UCDMO cross domain inventory version 3.4.0, June 2010. http://www.ucdmo.gov/ (accessed 23 Sept 2010).

[2] Jim Alves-Foss, W. Scott Harrison, Paul Oman, and Carol Taylor. The MILS architecture for high-assurance embedded systems. *International Journal of Embedded Systems*, 2(3-4):239–247, 2006.

[3] M. Anderson, C. North, J. Griffin, R. Milner, J. Yesberg, and K. Yiu. Starlight: Interactive link. In *ACSAC*, pages 55–64. IEEE Computer Society, 1996.

[4] M. Anderson, C. North, and K. Yiu. Towards countering the rise of the silicon trojan. Technical Report DSTO-TR-2220, DSTO Information Sciences Laboratory, 2008.

[5] Ross Anderson. *Security Engineering*. Wiley, first edition, 2001.

[6] D. E. Bell and L. J. LaPadula. Secure computer systems: mathematical foundations and model. Technical Report M74-244, The MITRE Corp., Bedford, Mass., May 1973.

[7] Department of Defense. Department of defense trusted computer system evaluation criteria, December 1985.

[8] DocuWiki Community. DocuWiki, 2010. http://www.docuwiki.org.

[9] D.A. Grove, T.C. Murray, C.A. Owen, C.J. North, J.A. Jones, M.R. Beaumont, and B.D. Hopkins. An overview of the Annex system. In *Proc. Annual Computer Security Applications Conference*. IEEE, December 2007.

[10] D.A. Grove, C.J. North, A.P. Murray, T. Newby, M.R. Beaumont, M. Chase, S. Haggett, and C.A. Owen. The Annex Multi-Level Secure computing architecture. DSTO technical report, 2011.

[11] Joshua P. Macdonald. File system support for delta compression. Master's thesis, University of California at Berkeley, 2000.

[12] Dylan McNamee, CDR Scot Heller, and Dave Huf. Building multilevel secure web services-based components for the global information grid. *CrossTalk*, pages 15–19, May 2006.

[13] T. Newby, D.A. Grove, A.P. Murray, C.A. Owen, and C.J. North. The second generation Annex TCB. DSTO technical report, 2011.

[14] Kar Leong Ong, Thuy Nguyen, and Cynthia Irvine. Implementation of a multilevel wiki for cross-domain collaboration. Technical report, Naval Postgraduate School, 2008.

[15] Colin Percival. Naive differences of executable code. http://www.daemonology.net/bsdiff, 2003.

[16] Neetu Rajpal. Using the XML diff and patch tool in your applications, August 2002. http://msdn.microsoft.com/en-us/library/aa302294.aspx.

[17] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[18] J.A. Rome and J.S. Tolliver. Multilevel architectures for electronic document retrieval. In *National information systems security conference*. Oak Ridge National Lab., TN (United States), 1997.

[19] Sebastian Rönnau, Geraint Philipp, and Uwe M. Borghoff. Efficient change control of XML documents. In *DocEng '09: Proceedings of the 9th ACM symposium on Document engineering*, pages 3–12, New York, NY, USA, 2009. ACM.

[20] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423,623–656, 1948.

[21] S. P. Shieh and V. D. Gligor. Detecting illicit leakage of information in operating systems. *Journal of Computer Security*, 4:123–148, 1996.

[22] Chengzheng Sun, Steven Xia, David Sun, David Chen, Haifeng Shen, and Wentong Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Trans. Comput.-Hum. Interact.*, 13(4):531–582, 2006.

[23] W. Mark Vanfleet, R. William Beckwith, Ben Calloni, Jahn A. Luke, Carol Taylor, and Gordin Uchenick. MILS: Architecture for high-assurance embedded computing. *The Journal of Defense Software Engineering*, August 2005.