# Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies

Lenin Singaravelu[1] Calton Pu[1] Hermann Härtig[2] Christian Helmuth[2]

[1]CERCS,
Georgia Institute of Technology
Atlanta, USA.
{lenin, calton}@cc.gatech.edu

[2] Technische Universität Dresden,
Institute for System Architecture.
Dresden, Germany.
{haertig, helmuth}@os.inf.tu-dresden.de

*The future of digital systems is complexity, and complexity is the worst enemy of security.* -- Bruce Schneier [40].

## ABSTRACT

The large size and high complexity of security-sensitive applications and systems software is a primary cause for their poor testability and high vulnerability. One approach to alleviate this problem is to extract the security-sensitive parts of application and systems software, thereby reducing the size and complexity of software that needs to be trusted. At the system software level, we use the Nizza architecture which relies on a kernelized trusted computing base (TCB) and on the reuse of legacy code using trusted wrappers to minimize the size of the TCB. At the application level, we extract the security-sensitive portions of an already existing application into an AppCore. The AppCore is executed as a trusted process in the Nizza architecture while the rest of the application executes on a virtualized, untrusted legacy operating system. In three case studies of real-world applications (e-commerce transaction client, VPN gateway and digital signatures in an e-mail client), we achieved a considerable reduction in code size and complexity. In contrast to the few hundred thousand lines of current application software code running on millions of lines of systems software code, we have AppCores with tens of thousands of lines of code running on a hundred thousand lines of systems software code. We also show the performance penalty of AppCores to be modest (a few percent) compared to current software.

## Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; D.4.6 [**Operating Systems**]: Security and Protection.

## General Terms

Reliability, Security.


*Eurosys'06*, April 18–21, 2006, Leuven, Belgium.


## Keywords

Application security, trusted computing base.

## 1. INTRODUCTION

Security-sensitive applications such as web browsers have grown tremendously in functionality and size. For example, the Mozilla browser contains 1 million lines of code. Browsers are used for many applications such as carrying out e-commerce transactions (e.g., handling credit card information), and viewing and updating personal information in bank accounts. Unfortunately, the growing code size has resulted in an increasing number of vulnerabilities. Attackers have successfully exploited these vulnerabilities to obtain private information or install arbitrary code that modifies the browser [11].

At the system level, libraries, middleware and kernel also have grown similarly in functionality and size. For example, the X11 window server contains over 1.25 million lines of code. X11 executes with superuser privileges and it has been vulnerable to buffer overflow exploits in the past [6]. A minimal functional configuration of the Linux kernel contains about 200,000 lines of code, and the whole kernel runs in privileged mode (x86 architecture). The Linux kernel too suffers from a host of vulnerabilities [7] including buffer overflow, privilege escalation and security bypass.

One approach to alleviate this problem is to extract security-sensitive parts of application and systems software, thereby reducing the size and complexity of software that needs to be trusted. Building small and simple software has been long advocated, e.g., Saltzer and Schroeder [38], in 1974, advocated *Economy of mechanism, Least privilege* and *Separation of privilege* as important design principles. Software engineering studies have also shown a positive correlation between software complexity and bugs in code [21][43]. In addition to the increased number of bugs, Schneier [40] argues that increased complexity also hinders the ability to understand and model the system, which leads to more difficult testing and analysis stages. Chen et al. argue that it would be more secure to run applications on virtualized machines than real machines, as a virtual machine monitor is considerably smaller and simpler than a regular operating system kernel [15].

The main contribution of this paper is an integrated approach to reduce the size of the trusted portion of the system. At the systems software level, we use the Nizza architecture as our trusted computing base (TCB). Nizza is a security architecture that relies on a kernelized TCB and on the reuse of legacy code – including whole operating systems – using trusted wrappers to control the size of the TCB. At the application software level, we extract the security-sensitive portions of an existing application into an AppCore. The AppCore is executed as a trusted process on the Nizza architecture and the rest of the application executes on a virtualized, untrusted legacy operating system.

We demonstrate the feasibility of this approach by implementing AppCores and TCBs for three real-world security-sensitive applications (e-commerce transaction client, VPN gateway and signatures in an email client). Compared to the few hundred thousand lines of code of current application software running on millions of lines of systems software code, the AppCore's tens of thousands lines of code running on a hundred thousand lines of system code are expected to be amenable to exhaustive testing or formal verification methods. We also show the performance penalty of AppCores to be modest (a few percent) compared to current software.

The rest of the paper is organized as follows. Section 2 describes the Nizza architecture, its design principles and main components. Section 3 describes the construction of AppCores and TCBs for the three security-sensitive applications. Section 4 discusses the security properties and shows the measured results for code size, complexity and performance of the resultant systems. Section 5 describes our experiences and observations regarding the construction of the AppCores and their TCB's. Section 6 discusses the related work and Section 7 concludes the paper.

## 2. NIZZA ARCHITECTURE

Nizza is a design for a small, secure and general-purpose platform supporting applications with high security requirements such as digital signatures and banking protocols while preserving the support for legacy code [23][24].

### 2.1 Design Principles

The Nizza architecture relies on three design principles: (1) build TCB out of small, isolated components, (2) use trusted wrappers to reuse untrusted components and (3) support legacy operating systems and applications.

Security-sensitive portions of many applications account for only a small fraction of the overall application complexity. As it is generally acknowledged that susceptibility to errors and attacks increases with complexity, the vulnerability of the security-sensitive part can be decreased significantly by isolating this part from the security-insensitive part. This leads us to the first design principle: Only essential security-sensitive functions should be part of the TCB. In other words, the TCB comprises only of components that cannot be omitted without compromising the functionality and security of the service. The security requirements fall into four main categories: confidentiality, integrity, recoverability, and availability. For clarity, we present the definition of these terms.

**Confidentiality:** Only authorized users (entities, principals, etc.) can access information (data, programs, etc.).

**Integrity:** Either information is current, correct, and complete, or it is possible to detect that these properties do not hold.

**Recoverability:** Information that has been damaged can be recovered eventually.

**Availability:** Data is available when and where an authorized user needs it.

We limit our work in this paper to confidentiality and integrity and Nizza is evaluated only on these two requirements. For many trusted components, data confidentiality and integrity are vastly more important than availability. Therefore, it is acceptable to use untrusted components if higher layers can guarantee the confidentiality and integrity of data. Trusted wrappers are components that help us achieve security objectives even when untrusted components are used. For example, the Secure Sockets Layer (SSL) library is a trusted wrapper for untrusted networks, as it allows us to use untrusted components like the Internet and an untrusted network stack, without compromising the confidentiality and integrity requirements of the application. Trusted wrappers enable the use of untrusted components like device drivers and network-protocol stack, allowing us to reduce the size of the trusted portion of the system.

To provide the full functionality of a standard OS, Nizza provides containers to securely run untrusted legacy OS components or even complete legacy OSes with their applications. Nizza facilitates cooperation among security-sensitive and untrusted components; Legacy applications can use an appropriately designed interface to communicate with trusted components, and trusted software can reuse legacy components through trusted wrappers.

### 2.2 Overview of the Nizza Architecture

Figure 1 shows a sketch of the Nizza architecture. Nizza is composed of four major parts: a small kernel; an execution environment consisting of trusted components (such as a name-server and window manager); an untrusted legacy OS with its applications; and security-sensitive applications (discussed in Section 3). In all figures, shaded boxes represent trusted components and plain boxes represent untrusted components.

#### *2.2.1 Kernel*

The basic requirements for the small kernel are that it enforces component isolation in protection domains and provides fast communication between these domains (re-

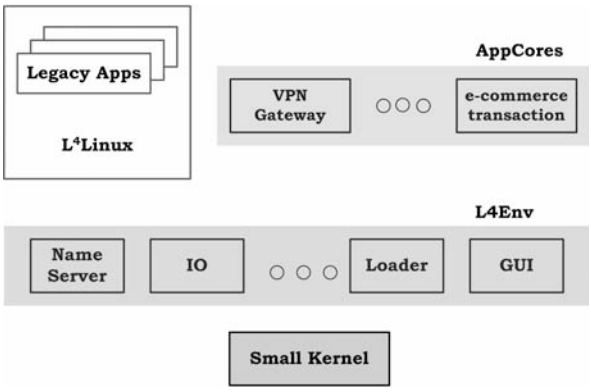


**Figure 1. Overview of the Nizza Architecture. Shaded boxes represent trusted components.**

quired for trusted wrappers and other secure-platform components). The Nizza architecture is based on the L4 microkernel [32]. The L4 microkernel provides three abstractions – threads, address spaces and IPCs. Components are executed as L4 threads and isolation is enforced via address space separation. IPC calls provide a fast communication channel to transfer control, data or memory pages.

The current L4 interface has some restrictions that must be overcome for a complete Nizza implementation: It currently lacks kernel-resource control and IPC control. Thus, untrusted components cannot be contained completely. These problems are expected to be fixed in future versions of the kernel.

#### *2.2.2 Execution Environment*

System servers running in the trusted portion provide services that are essential to the functioning of the system. These servers form the trusted execution environment for the secure applications. We refer to the execution environment of L4 as *L4Env*.

The composition of execution environment varies depending on the needs of the application. The base configuration of L4Env contains a name server and resource managers for main memory, CPU, and I/O. In addition to these essential services, system server's specific to each application scenario may be incorporated into the TCB. The window manager (GUI) is an example of an optional component that provides an interface to manage the display. Additionally, the window manager provides an unforgeable trust indicator to the user. The current implementation of the window manager uses a few pixels at the top of the screen, which cannot be accessed by untrusted applications, to display the authentication chain of the in-focus application. The window manager also dims windows that are not in focus to provide an unambiguous association between the active window and the authentication chain [19]. The *loader* is an optional dynamic loader and linker responsible for loading new components at runtime. The loader needs to be aware of authenticated booting. It establishes the authentication chain and makes it available to other trusted components when necessary (e.g., to the window manager).

#### *2.2.3 Support for Legacy Code*

Support for a legacy OS and its applications is critical for the acceptance of a new architecture. The current implementation of Nizza architecture supports L$^4$Linux [25], a paravirtualized Linux kernel. L$^4$Linux is binary compatible with the unmodified Linux kernel and this allows the reuse of existing Linux applications. The modified kernel is executed as a user level task and is isolated by the microkernel. Therefore, it cannot harm the trusted components, even if its core, the former Linux kernel, is compromised.

### 2.3 Hardware Requirements for Nizza

Nizza relies on hardware support as described in the specifications of the Trusted Computing Group [10] to support the security properties described above. Specifically, it relies on authenticated booting to establish a chain of authentication for the executing software. The authentication chain can be used to reassure a remote party about the application being executed (*remote attestation*). Locally, a user can compare the chain of trust with a secure copy (e.g., a copy on a USB key) before logging into the system. Once logged in, the window manger uses the chain of trust to inform the user about the trustworthiness of the in-focus application. *Sealed storage* protects the confidentiality of the cryptographic keys that are foundations for the security of the rest of the system.

## 3. CASE STUDIES IN CONSTRUCTING SECURITY-SENSITIVE APPLICATIONS

Applications that perform security-sensitive tasks or handle security-sensitive data have to be trusted. Therefore, these applications should be as small and simple as possible, as long as they satisfy the functionality and security requirements. The large size of current application software code (e.g., 1 million lines of code in a browser) makes straightforward porting of existing applications an unattractive solution. Section 3.1 presents our solution of extracting security-sensitive functionality of existing applications into a small AppCore. The AppCore is then executed as a trusted process, while the rest of the application executes as an untrusted process.

### 3.1 Constructing AppCores

The process of extracting an AppCore from an existing application can be broadly divided into three stages: (1) analysis of the application to identify security-sensitive components, (2) extracting the identified components and composing them into an AppCore and (3) modifying the original application to use the AppCore for security sensitive tasks.

The function of the analysis stage is to identify components that handle security-sensitive data or perform security-sensitive functions. If the application has reasonable documentation, this step can be accomplished by

analyzing the documentation, e.g., the Mozilla browser for the E-Commerce transaction client scenario (Section 3.2) is well documented and has clearly-defined modules making identification easier. Otherwise, we have to manually identify security-sensitive functions based on domain knowledge, as in the case of the VPN gateway AppCore (Section 3.2). This stage can be partially automated by using dataflow analysis, as described in [14].

In the next stage, we extract the security-sensitive components and integrate them into a standalone AppCore. There are two factors that control component integration: First, to the greatest extent possible, we want to reuse the interfaces between the security-sensitive components and the rest of the application, and second, we want to constrain the security-sensitive components to perform only the requisite security-sensitive tasks. Reusing existing interfaces simplifies the reintegration of the AppCore with the application. Constraining security-sensitive components involves modifying or rewriting components to perform the necessary security-sensitive task with least amount of software, e.g., substituting Mozilla's NSS module with a bare-bones SSL library, MatrixSSL, provides size savings over two orders of magnitude. However, this could also break existing interfaces and increase the cost of reintegration. Thus we have two conflicting factors influencing component integration. The design choices are discussed in greater detail in Section 5.

The final stage consists of going through the components in the original application and replacing the existing function calls to security-sensitive modules with calls to the new AppCore. In our case studies, this turned out to be a straightforward process as we were either reusing interfaces or the application had a plugin architecture and we connect the application to the AppCore via plugins.

We study three distinct applications: an e-commerce transaction client, a VPN gateway implementation and an email signer. The applications provide variety in terms of security properties and implementation complexity. The e-commerce transaction client protects the confidentiality and integrity of user data, the VPN gateway software protects the confidentiality and integrity of the private network's data and the email signer provides integrity of email content. The e-commerce transaction client and the email signer are standalone applications that require analysis of a single software program. On the other hand, the VPN gateway implementation requires analysis of a multi-level software stack, including the operating system kernel (network protocol stack) and the security library.

## 3.2 E-Commerce Transaction Client

The most popular tool for carrying out an e-commerce transaction is a browser. Browsers perform two critical functions for e-commerce – they display content, in a format determined by the merchant, to the user and they accept user input and pass them along to the merchant. Data transfers can be protected using a transport layer security protocol like SSL or TLS. In a typical e-commerce transaction, initially, the customer builds up a shopping cart, which can involve multiple rounds of merchant-customer interaction. Next, the customer decides to finalize the transaction. At this point most merchants use a transport layer security protocol to protect any further communication. Once a secure layer has been established, the customer provides the merchant with payment information or unique login information to retrieve a profile. The merchant verifies this information and finalizes the transaction.

The large code base of browsers and the support for extensions via tightly integrated (i.e. executing in the same address-space) plugins hinder effective testing of browsers. Browsers are sources of multiple vulnerabilities including arbitrary code execution and security bypass [4][5]. Browsers also suffer from spoofing vulnerabilities where the attacker is able to fool the user into mistaking an arbitrary site for a trusted site. Attackers have successfully exploited these vulnerabilities to install malicious plugins, and steal private information like passwords and credit card information. These vulnerabilities illustrate the risk in using a browser to carry out security-sensitive operations like online purchases. On the other hand, since a majority of merchants and consumers prefer to use the browser as a transaction tool, an effective solution must work within the framework of the browser.

**Table 1: Modules in the Mozilla Browser [2]**

| Type | Example Modules |
|---|---|
| Main Browser | Browser, Portable Runtime, Display Widgets, New HTML Parser. |
| Security | Security (NSS & JSS). |
| Scripting | Javascript Engine, Rhino, Live Connect. |
| Security-Extras | Personal Security Manager, JS Security. |
| UI-Enhancements | Clipping & Compositing, Find as you type, ImageLib, accessibility. |
| Parsing-Extras | RDF, DOM, XML, XSLT, MathML. |
| Extras | I18N, URI Loader, Zlib, Qt support, Cookies, Plugins, Preferences, Update. |

The security-sensitive data in an e-commerce transaction client is the user's payment and shipping information, the shopping cart that is displayed to the user, and the user's choice about the transaction. Upon analysis of the browser's components (Table 1), we find that the following components are security-sensitive: the main browser, which contains the parser, the display, the security library and basic user-interface components. These components form a significant portion (over 50%, 500 KLOC) of the browser's code base. Composing them into an AppCore would result in a trusted application smaller than the original browser, but we can achieve better results by refining the selected components. We simplify the selected components by limiting their functionality. For example, constraining the language describing the shopping cart would allow for a smaller parser and display

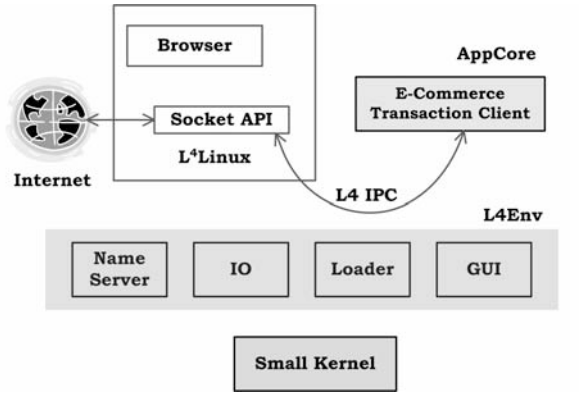


**Figure 2. AppCore and TCB for a Transaction Client**

interface. Similarly, implementing only the most used cryptographic mechanisms would significantly reduce the size of the security library. We now consider these components in more detail.

**Browser Parser.** This browser component has to understand a variety of HTML and XML standards with multiple variants. A shopping cart, for example, can be described in detail using sophisticated tags and templates. While this is an expected good usability feature, it introduces potential vulnerabilities due to growing code size. A cart described using the table tags in HTML would require a small and specialized parser. As shown in Section 4.2, this specialization helps us achieve significant reductions in the code size and complexity of the AppCore.

**User Interface.** There are many demands on a generic web browser; their functionality has to include the display of text and images in multiple formats, support novelties such as tabbed browsing and skins, etc. In contrast, a small interface designed for a shopping cart would focus on the unambiguous display of the cart content and a clear way to accept or decline the transaction. Our AppCore uses a text-box-based interface that displays the table and presents the user with a confirmation window. We were able to implement this functionality with less than 500 LOC. While our current interface is primitive, given the limited expressiveness of the cart data, we expect that even a more sophisticated interface would be orders of magnitude smaller than a generic browser.

**Security Library.** A library implementing either the SSL or TLS protocol is necessary to carry out secure transactions over the Internet. In our implementation, we use MatrixSSL [3], an SSL library with a small footprint originally developed for embedded systems. The library provides a minimal set of standard algorithms necessary for SSL. At less than 10 KLOC it is significantly smaller and less complex than SSL library in Mozilla (NSS module in Mozilla is over 180 KLOC).

**AppCore.** The parser, security library and the user interface components are put together to form the AppCore of the e-commerce transaction client. The overall system, depicted in Figure 2, consists of the AppCore, labeled as E-Commerce Transaction Client, the microkernel and an execution environment with the basic L4Env and a trusted window manager (GUI). The AppCore relies on an untrusted socket interface proxy to communicate with the external network. This is a concrete example of the use of trusted wrappers: We use SSL in the AppCore to protect the confidentiality and integrity of the data before passing it over to the proxy. Hence we can afford to use an untrusted network stack for communication.

## 3.3 VPN Gateway

A Virtual Private Network (VPN) is a private network that uses public networks for communication. VPN gateways are used to provide confidentiality, authenticity and integrity for private network data. These gateways behave like trusted wrappers, using a tunneling protocol like IP Security Protocol (IPSec), Point to Point Tunneling Protocol (PPTP) or Layer 2 Tunnel Protocol (L2TP) to satisfy the security requirements. In IPSec-based VPNs, all messages are encrypted using IPSec-compliant tools before sending them over public networks (e.g., the Internet). Therefore, the VPN gateway acts as the guard at the border between networks of different trust/security levels.

The majority of current VPN implementations are based on monolithic operating systems (e.g., Linux). In this case, the IPSec implementation is integrated with the kernel and intertwined with the network subsystem. Consequently, vulnerabilities in the kernel, the network subsystem or security library can be exploited to compromise the VPN software. Some of the vulnerabilities include buffer overflow [8], which can be used to gain control of the VPN system, and information leak vulnerabilities, which allow an attacker access to sensitive information.

A software based VPN implementation generally contains an operating-system kernel, possibly a stripped down version (e.g., Snapgear's Embedded Linux), and an implementation of the tunneling protocol (e.g., FreeS/WAN library provides an IPSec implementation). We discovered that the security-relevant functions of a VPN gateway implementation – data protection and policy enforcement – comprise only a small fraction of a monolithic kernel (less than 5%). We analyzed the FreeS/WAN implementation for Linux to identify the data protection and policy enforcement functions, which were then extracted into a AppCore called Viaduct.

**AppCore.** Figure 3(a) illustrates our initial implementation of the VPN gateway called Mikro-SINA, which is described in detail in an earlier paper [26]. Mikro-SINA consists of three main components: On the right is the *private code* that interacts with the private network. On the left is the *public code* that interacts with the Internet. In our initial implementation, the private and public network code each use a separate, fully featured L$^4$Linux VM. Interactions between the two VMs are mediated by Viaduct. To ensure confidentiality and integrity of private data, the Viaduct encrypts all traffic before passing it to

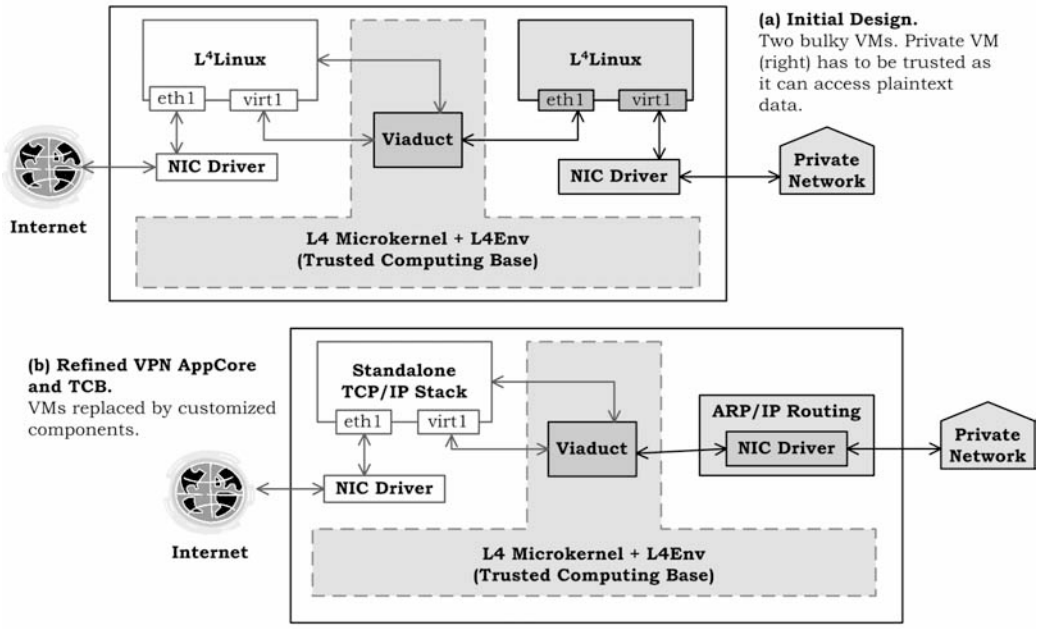


**Figure 3. Mikro-SINA. A VPN Gateway implementation on the Nizza Architecture.** Viaduct is the AppCore proper. The VM on the right, in (a), and ARP/IP Routing software, in (b), are also trusted components.

the public L4Linux VM. Conversely, the Viaduct decrypts the data it receives from the public L4Linux VM and verifies the integrity of the data before passing it to the private L4Linux VM. Therefore, the untrusted public code can only access encrypted data (IPSec ensures confidentiality and integrity of data). Although the private VM cannot leak information through Viaduct, it can see plaintext data and therefore has to be trusted as much as the rest of code on the private side. Hence, the size of the private L4Linux VM is an added concern.

In an effort to further improve the security properties of the Mikro-SINA VPN, we refined the private L4Linux VM. Instead of the full L4Linux VM, we now use customized, small components that only perform the functions required for a fully functional VPN gateway. The result is shown in Figure 3(b). The private side L4Linux is replaced by the NIC driver plus an address resolution protocol (ARP) that translates IP addresses into link layer addresses (e.g., Ethernet MAC address). The public side L4Linux is replaced by a standalone TCP/IP stack implementation. The TCP/IP implementation is a port of the Linux 2.4 network stack and it runs directly on top of the trusted platform as an untrusted process.

## 3.4 Signatures in an Email Client – Enigmail

Email signing is used to protect the integrity of the e-mail content. When a user wants to sign an email, he activates an extension, which reads the user's private key and signs the email's content. In some cases, the private key may be in encrypted format, and the user must provide a passphrase to decrypt the key. The content and the signature can now be sent over an unprotected network.

Mozilla Thunderbird is a standalone email client that supports email signing via a third-party plugin – Enigmail. Enigmail uses the GnuPG library to read the user's private key and sign emails. Enigmail relies on Thunderbird to communicate with the user and maintain passphrases. Effectively, Enigmail has to rely on the correctness of the email client and the multitudes of plugins that can be installed on the Thunderbird client to perform its security-relevant functions correctly. Thunderbird alone contains over 200 KLOC, making it a very difficult task to ensure correctness of the email client. Thunderbird also shares some of the libraries with the Firefox browser (e.g., HTML parser), thereby sharing the vulnerabilities too.

**Table 2: Modules in Mozilla Thunderbird**

| Type | Example Modules |
|---|---|
| User Interface | Folder view, Email compose |
| Account Mgt. | Address book, Mail account controls |
| Extras | Spell checking, Junk mail control, Message filters, Themes, Software Update |
| Security | Password manager, Certificate manager, Enigmail (as plugin) |

To sign an email in Thunderbird, the user selects the signing option, composes the email and clicks the send button. The Enigmail plugin then retrieves the passphrase for the user's private key and sends it along with the contents of the email to the GnuPG library. GnuPG retrieves the key, signs the email and sends it back. The security requirements for the Enigmail plugin are confidentiality and integrity for the user's private key and passphrase and integrity of the content of the email that is signed. Table 2 presents a list of identifiable modules in Thunderbird. The security modules along with the user interface modules need to be extracted to satisfy the security requirements.

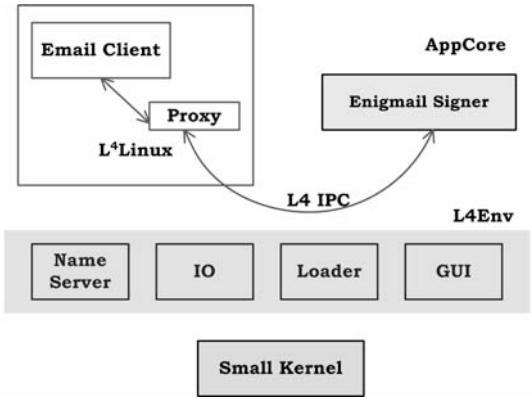


**Figure 4. AppCore for Email Client**

**AppCore.** The AppCore must be capable of performing two functions: First, display the content of the email in an unambiguous fashion, and second, depending on the user's preference, sign the email with the user's private key. It is important to note that plaintext content is available to the untrusted component even before signing. It is the user's responsibility to ensure that the content has not changed when the untrusted application passes it to the AppCore. Since the content is signed in the trusted part, the key management functionality (GnuPG library) must be included in the AppCore. The Enigmail AppCore displays the textual content of the email and waits for user input. Depending on the preference and the passphrase provided by the user, the email is either signed and sent back to the client or discarded. Currently, Enigmail's AppCore is limited to displaying text, but it is possible to add display modules for other standard attachments.

# 4. EVALUATION

We implemented the AppCores for the three applications on the Nizza architecture (Section 2). The AppCores are executed as trusted processes, directly on top of the microkernel. The untrusted components of applications run on top of L$^4$Linux, a virtualized, untrusted operating system. The TCB for each application consists of the L4 microkernel and a set of basic resource managers. In the e-commerce transaction and the Enigmail scenarios, the TCB also includes a window manager (GUI).

We choose a trusted computing base based on the Linux kernel as a competing implementation as we have unencumbered access to its source code. Moreover, the Linux platform has open-source implementations for all three applications. The trusted computing base must contain the Linux kernel. In the case of the e-commerce transaction client and the email client, the X Server is also a part of the trusted computing base.

Section 4.1 discusses the impact of AppCores on the security of the system. In Section 4.2, we show that it is possible to construct TCBs for real-world applications within 100,000 LOC. Section 4.3 discusses the performance penalty incurred by using AppCores instead of the original application and we show that the performance drop is modest in most cases.

## 4.1 Security Properties of AppCores

The main focus of the AppCore research is to reduce the size of the trusted computing base at the operating system, middleware and the application layers. A smaller and simpler code base has two advantages – a smaller code base for testing and analysis and a smaller attack profile. A smaller code base eases the software assurance process and Section 4.2 discusses the reductions achieved in greater detail. The lack of an extension architecture is a concrete example for attack profile reduction in the e-commerce transaction client and the Enigmail client. Vulnerabilities in the extension architecture and malicious extensions have been used to compromise the confidentiality and integrity of content [11]. Since extensions are extraneous to security requirements in our case studies, the resultant AppCores do not possess an extension architecture. Note that the extension architecture is missing only in the trusted part of the application. The original application retains its extension architecture; therefore, we are able to improve security without sacrificing user experience.

The Nizza architecture provides confidentiality and integrity of data at the operating system and middleware level. These properties are extended to the application layer by the AppCores. In the e-commerce transaction client and the VPN gateway scenarios, the AppCores provide confidentiality and integrity of data by using a transport layer security protocol like SSL or IPSec to encrypt data before handing it over to untrusted components. The Enigmail scenario requires integrity of email content, which is enforced by signing the email with the user's private key before passing the content and the signature to the untrusted application.

Recall that the Nizza architecture provides us with a trusted window manager that controls the top portion of the screen (Section 2.2.2). The top of the screen, which acts as an unforgeable trust indicator, is used by the window manager to indicate the trust level of the in-focus application. We assume that the users are a part of the TCB, i.e., they are aware of the trust indicator when they give out sensitive data or perform security-sensitive operations. Since, untrusted applications cannot modify the trust indicator, interface spoofing attacks are minimized.

It is important to note that AppCores do not address the issue of availability, e.g., malicious entities could carry out denial of service attacks by corrupting content, hijacking the untrusted applications, or dropping packets carrying content from, or to the trusted applications. While AppCores can detect these attacks, or infer their presence due to degradation of service, they cannot protect the application against such attacks. Defensive programming techniques [34] can be used to make AppCores more resistant to these types of attacks.

## 4.2 Software Complexity Metrics

Given the expected order-of-magnitude differences in code size, the comparison between an application and its

AppCore do not require a very precise metric. Therefore, we chose well known software complexity metrics - Source Lines of Code (LOC), measured using SLOCCount [48], and McCabe's complexity metric (MCC) [33]. Our goal is to show the overwhelming advantage of AppCores from many angles, not to distinguish fine differences.

Many software engineering studies have shown that the LOC metric is roughly correlated with the number of defects [21][43], which may be relatively imprecise, but serves as a useful baseline measure [13]. MCC is based on McCabe's definition [33] of a control flow complexity metric. It is measured per function and it gives the number of distinct execution paths in a given function. Intuitively, it represents the minimum number of tests that need to be carried out on that function to verify control flow properties. One of the weaknesses associated with the MCC metric is its omission of call nesting depth [44]. So, we also include measurements of call depth as a separate parameter in our evaluation.

We are aware that traditional software complexity metrics like MCC and SLOC have their disadvantages, e.g., their dependence on programming language and style [20]. Some of the original applications have had post-release periods in terms of years allowing for multiple bug fixes, whereas our approach will result in relatively new AppCores. Our argument is that vulnerabilities are still being consistently found in those applications [4][5]. We show that AppCores are considerably smaller and simpler, providing us with a clear advantage in testing.

#### *4.2.1 E-Commerce Transaction Client AppCore*

Before we compare the code size, we briefly discuss the functional equivalence between the two programs (our e-commerce transaction client AppCore and a generic web browser such as Mozilla) as well as their security goals. Both approaches support a shopping cart-based electronic commerce application, with secure data transmission between the client and server typical of such security-sensitive applications. The data confidentiality and integrity requirements are fulfilled by a secure transport protocol such as SSL. Table 3 shows the contrast between a generic browser-based application and an AppCore-based approach. We compare the software complexity (in LOC, MCC, and call depth) of the security-sensitive code from the two approaches. The AppCore simplifies the functionality of the security-sensitive code in two main components: a smaller SSL library and a very narrow subset of HTML to describe the cart. We observe the simplification of components provides considerable reductions in software size. The entire AppCore is about one hundredth the size of the Mozilla browser. Similar reductions are seen with the MCC metric.

**Table 3: Complexity Comparison of E-Commerce Transaction Client AppCore and Mozilla.** Level of shading indicates functional equivalence of components.

| Component | LOC | Cumul. MCC | Avg. MCC | Avg. Depth |
|---|---|---|---|---|
| **MatrixSSL** | 8,600 | 1,200 | 7.5 | 1.70 |
| **Mozilla-NSS** | 180,000 | 24,700 | 8.7 | 1.47 |
| **Custom Parser** | 200 | 35 | 5.8 | 1.70 |
| **HTML-Mozilla** | 19,000 | 3,100 | 15.3 | 2.08 |
| **AppCore** | 10,000 | 1,500 | 7.4 | 1.67 |
| **Mozilla** | 978,000 | 151,000 | 6.2 | 1.72 |

#### *4.2.2 VPN Gateway AppCore*

The initial Mikro-SINA VPN design (Figure 3a) that uses L$^4$Linux for the private network code results in a system with a large code base that needs to be trusted (~ 200 KLOC for L$^4$Linux). The minimalized Mikro-SINA (Figure 3b) resolves the problem by refining the trusted processes, replacing L$^4$Linux (on both private and public sides) with customized network stack. The result is a very small VPN gateway, both in terms of trusted AppCore and a relatively small untrusted networking code, with about 80 KLOC on the public side, and a small ARP component (8000 lines) on the private side. Table 4 compares the software complexity of Mikro-SINA to the FreeS/WAN implementation. The Linux kernel numbers are obtained from Snapgear's Embedded Linux distribution [9]. The Mikro-SINA VPN possesses a twofold advantage in complexity and size over the FreeS/WAN implementation.

#### *4.2.3 Enigmail AppCore*

The original Enigmail plugin is tightly integrated with the Thunderbird email client. It also relies on the GnuPG library to sign and verify emails. The AppCore for Enigmail has its own display and user input modules but it still requires the GnuPG library to sign emails. Thunderbird, Enigmail and the GnuPG library together account for over 250 KLOC, whereas the Enigmail AppCore which includes the ported GnuPG library contains less than 54 KLOC. The results for MCC metric show a fourfold reduction in complexity – 11,000 for the AppCore and 45,000 for the Thunderbird application along with the GnuPG library.

**Table 4: Complexity Comparison of Mikro-SINA and FreeS/WAN**

| Component | LOC | Cumul. MCC | Avg. MCC | Avg. Depth |
|---|---|---|---|---|
| **Mikro-SINA Viaduct** | 10,400 | 990 | 4.1 | 1.31 |
| **FreeS/WAN** | 34,100 | 4,300 | 7.9 | 1.56 |
| **Mikro-SINA VPN + L4+L4Env+Viaduct** | 74,000 | 10,000 | 4.6 | 1.54 |
| **FreeS/WAN + Snapgear Linux** | 155,000 | 25,000 | 5.8 | 1.59 |

**Table 5: Complexity of the Trusted Computing Base**

| Component | LOC | Cumul. MCC | Avg. MCC | Avg. Depth |
|---|---|---|---|---|
| **L4 microkernel** | 14,000 | 2,300 | 3.5 | 1.39 |
| **Basic L4Env** | 54,500 | 7,000 | 5.0 | 1.58 |
| **GUI** | 35,600 | 4,600 | 6.7 | 1.36 |
| **Loader** | 37,000 | 5,000 | 5.7 | 1.69 |
| **Basic TCB: microkernel + Basic L4Env** | 69,000 | 9,500 | 4.5 | 1.54 |
| **Basic TCB + GUI** | 87,600 | 12,300 | 5.2 | 1.48 |
| **Basic TCB + GUI + Loader** | 100,200 | 14,000 | 5.2 | 1.50 |
| **Linux Kernel + XServer** | 1,485,000 | 238,000 | 7.7 | 1.73 |

#### *4.2.4 Software Complexity of a TCB*

We have seen that at the application level, refactoring and refining an application significantly reduces code size and complexity. Since the AppCores will be running on top of a TCB, it is also necessary for the TCB to be relatively small and simple. Using a Linux kernel (over 200 KLOC) as a TCB would run counter to the notion of a small TCB.

The composition of the TCB varies depending on the requirements of the AppCore. The microkernel is an integral part of our TCB. The basic L4Env consists of resource managers, naming service and an IO server. The window manager (GUI) and loader are optional components. An AppCore like Mikro-SINA would require just the basic configuration. AppCores like trusted email signer and transaction clients would require a GUI enabled TCB. A dynamic loader is required for situations where a user or application would like to load new trusted applications. Table 5 lists the complexities of the various TCB components and configurations. Since some files are shared between various servers, each progressively complex TCB configuration does not increase proportionally in size and complexity. The first observation from the measurements is that significant size and complexity reductions can be attained with careful composition of the TCB. Secondly our TCB configuration is an order of magnitude smaller than a Linux based platform.

### 4.3 Performance

Our approach of refactoring applications generates two processes coordinating to perform a task that previously required a single process. While refactoring improves security, as now a smaller portion of the original application has access to sensitive data, it also results in performance degradation. There are two main contributors to the performance penalty: first, the overhead of running the application on top of a virtual machine and second, data transfer and context switch times between the AppCore and the application. The issue of application performance on L$^4$Linux has been addressed in detail in an earlier work [25]. The conclusion was that performance penalty for most applications can be contained within the 5-10% range.

We present the results of a series of micro-benchmarks to demonstrate the performance of the main L4 - L$^4$Linux interactions.

#### *4.3.1 Data Transfer Rates between L$^4$Linux & L4.*

The following experiments were performed on a Pentium-4 2.24 GHz machine with 512 MB RAM and front side bus speed of 400 MHz. The standard deviation for the experiments in this section is less than 1 % of the mean.

Each L$^4$Linux process is implemented as an L4 task [25]. Communication between an L4 process and an L$^4$Linux process uses L4's IPC mechanism, which is independent of the guest operating system (L$^4$Linux). Therefore we just have to measure communication latency and throughput between two L4 tasks. L4 provides multiple ways for inter-address space communication and data transfer. We discuss two of them - Direct IPC, which uses registers to transfer data and Indirect IPC, which uses string copies to transfer data.

**Table 6: 32 Bit Transfer: via registers**

| Client Timeout | Throughput (MBps) | Context Switch (us) |
|---|---|---|
| Inf. Timeout | 2.90 | 0.690 |
| 1 sec Timeout | 1.73 | 1.158 |

The results in Table 6 are for inter-address space direct IPC, in which 32 bits of data is transferred via registers. The client calls the server with a receive-timeout value and the server replies with a 32 bit value in a register. Since L4 optimizes the infinite-timeout-value (inf-timeout) data transfers, we present the results for inf-timeout and a nominal one-second-timeout data transfer. The context switch times are calculated from the throughput values (Inverting the throughput value and dividing by two). We can see that direct IPC is not suited for data transfers due to low throughput, but they can serve as an inexpensive notification mechanism.

L4 provides support for fast data transfers via indirect string transfer IPCs. In our experiments, the client calls the server with an inf-timeout value and the server replies with an indirect string, which is copied to the client's address space. Table 7 compares the throughput of indirect IPC against FIFO's in Linux. We see that the maximum throughput values in both systems are comparable.

**Table 7: Data Transfer Throughput in MBps. Linux FIFO vs. Indirect String Transfer in L4.**

| Size (Bytes) | FIFO (Linux) | Indirect String Transfer (L4) |
|---|---|---|
| 256 | 68.95 | 88.01 |
| 4,096 | 588.55 | 863.81 |
| 8,192 | 664.69 | 792.17 |
| 16,384 | 799.03 | 801.70 |
| 32,768 | 822.22 | 763.89 |
| 65,536 | 879.94 | 715.95 |

### *4.3.2 E-Commerce Transaction Client*

We now measure the time to carry out a typical e-commerce transaction using the transaction client AppCore. The transaction client initiates an SSL connection and receives the cart from the web-server. It displays the cart and waits for the user's response. Depending on the user's response, the client either sends the user's payment and shipping information back to the server and finalizes the transaction or aborts it. Since we are interested in finding out the minimum time to execute the transaction, we eliminate user-interaction by assuming that the user always wants to accept the transaction. Table 8 lists the server-side execution time for a transaction over a loopback interface. Each column title represents the execution environment of the client and server respectively. The execution time for the trusted scenario (L4 - L$^4$Linux) is around 11 % slower than the Linux scenario. But in absolute terms, the execution time is less than 50 ms, which is insignificant when compared to user response time (order of seconds).

**Table 8: Execution time for an E-Commerce Transaction**

| Client-Server pairs | Linux-Linux | L$^4$Linux - L$^4$Linux | L4 - L$^4$Linux |
|---|---|---|---|
| **Time (ms)** | 39.1 | 40.2 | 43.5 |
| **Stdev %** | 0.2 | 10.6 | 8.0 |

### *4.3.3 Mikro-SINA VPN Gateway*

Table 9 presents the throughput results from the Netperf TCP_STREAM benchmark for the competing VPN implementations. The experiment duration per test run was 1 minute. The VPN implementations ran on a Pentium-4 1.80 GHz machine with 512 MB RAM and two EEPRO Fast Ethernet enabled cards. In the FreeS/WAN implementation, a single process handles both the external and internal interfaces. This design minimizes overhead and allows FreeS/WAN to achieve higher throughput. On the other hand, Mikro-SINA separates the handling of the external and internal interfaces. The separation adversely affects the performance as shown in Table 9 but provides better security features as an attack on the external interface will not provide the attacker with access to plaintext data.

**Table 9: Throughput Comparison of Mikro-SINA and Free/SWAN.** All numbers are in Mbps.

| Encryption | Linux-2.4 FreeS/WAN | Mikro-SINA |
|---|---|---|
| **Null** | 94.9 | 93.9 |
| **3DES-MD5** | 63.8 | 32.2 |

## 5. LESSONS LEARNED

The construction and evaluation of AppCores and their TCBs presented us with some insights. Some of our observations are straightforward – AppCore construction is easier if we can establish a clean separation between the security-sensitive and security-insensitive portions. This is possible if (a) the original application has a well-defined and well-documented modular design and (b) the security-sensitive task spans a small subset of the modules. Other observations revealed themselves as crucial design-decisions during the construction process.

**Identifying the Point of Separation: Sacrificing performance for security**

Deciding on the point of separation is a straightforward process in many cases. For example, in the e-commerce transaction scenario, the point of separation occurs when the user has to input security-sensitive data (profile or payment information) and in the email client scenario, the point of separation occurs when the user has to enter a passphrase to retrieve his key.

However, there are cases where there is more than one point of separation – generally involving a tradeoff between performance and modularity. For example, in the FreeS/WAN implementation, IPSec policy enforcement occurs at the IP layer, whereas the actual encryption/decryption occurs after the TCP layer. This optimization improves performance as the IP layer can drop packets that do not confirm to the policy rather than dropping the errant packets after transport layer processing. But modularity is sacrificed as the security subsystem is now entangled with other functionality. During the construction of the VPN gateway AppCore, we had three choices (a) incorporate the network stack into the TCB (b) keep the network stack out of the TCB and replace IPSec function calls from the untrusted network stack with IPC calls to the trusted components or (c) keep the network stack out of the TCB and sacrifice the performance optimizations at the IP layer. The first choice was not really considered as it would mean increasing the size of the TCB by almost half (40,000 LOC). The second option resulted in a messy separation as there were trusted calls from multiple layers of the network stack (IP layer needs access to IPSec policy and TCP layer provides the packets for encryption/decryption). The third option sacrifices some performance by eliminating the optimizations from the network layer but results in a simpler interface between the trusted IPSec implementation and the untrusted network stack (send and receive packets).

**Extracting security-sensitive components: Reusing existing interfaces simplifies separation and reintegration**

Ideally, when we split an application into security-sensitive and security-insensitive parts, we would like to retain the interface between the two parts. Reusing existing interfaces allows us to limit the changes to modifying the call type, e.g., changing security-sensitive function calls to L4 IPC calls with appropriate marshalling and unmarshalling code. This involves identifying the location of the calls, which can be accomplished with tools in most cases. If the original application possesses a plugin architecture (e.g., plugin architecture in Mozilla Thunderbird), this task is even simpler, as we just have to write a plugin that acts as a bridge between the AppCore and the untrusted application.

**Refining security-sensitive components: Reducing complexity through manual analysis**

Once we have extracted security-sensitive components, we can achieve further reductions in size and complexity by refining the components to perform only the requisite security-sensitive tasks. This requires considerable manual effort; for each component that can be refined, we have to incorporate simpler alternatives. This often breaks the interfaces between the security-sensitive components but it is worth the effort as we gain enormous savings in terms of software size and complexity.

For example, we considered various alternatives when replacing the HTML parser from Mozilla (19K LOC) for the E-commerce transaction client. Our first step was to use the Expat XML parser, which contains about 10K lines of code. The use of XML allowed us to have rich, extensible language for the shopping cart. But it represented over 50% of the final AppCore. This led us to consider various alternatives for a shopping cart description language – trading off extensibility for size and complexity (plain text could be ambiguous for the user, compressed image formats require complex parsers, and raw images consume too much bandwidth). Finally, we settled on the *table* tags of HTML to describe a shopping cart. Since we changed the shopping-cart description format, we had to change the parser for the AppCore and the *remote agent* (usually a web-server) involved in the transaction. On the other hand, the custom parser provided limited extensibility and size and complexity savings over one order of magnitude (Table 3).

The initial design for the VPN gateway (Section 3.3) had a full-fledged Linux-based VM as a trusted component. While the underlying TCB prevents the private VM from communicating with untrusted components or the external network, we would have to rely on its correctness to preserve the confidentiality and integrity of plaintext data. Since the private VM is just used to transmit packets over the private network, we decided to replace it with ARP/IP routing software. While this required reprogramming the AppCore and getting the ARP code to run as a standalone process, we were able to replace the 200 KLOC of L$^4$Linux with 8 KLOC of ARP code.

**Trusted Wrappers: Tradeoffs between system complexity, security and performance**

Trusted wrappers are components that enable the utilization of untrusted components in lower layers of the system. We were able to reuse the network stack executing in an untrusted and virtualized legacy operating system as we used security protocols like SSL (E-commerce transaction scenario) and IPSec (VPN gateway scenario) to protect the confidentiality and integrity of data. The use of trusted wrappers allowed us to push the TCP/IP stack out of the TCB, which resulted in savings of over 40 KLOC. This approach also provides another significant advantage: we retain the full functionality of the original network stack implementation, which cannot be provided by alternate solutions like lightweight TCP/IP implementations.

Trusted wrappers do not come for free. We can look at trusted wrappers as exploring tradeoffs in two different areas. The first tradeoff deals with attacks: attacks on confidentiality and integrity on one hand and attacks on availability on the other. Trusted wrappers allow us to (re)use untrusted components. We are therefore exposing a trusted application to denial of service attacks via the reused untrusted components. On the other hand, we are also reducing the attack profile for attacks on confidentiality and integrity by reducing the size of the trusted application. This is an acceptable tradeoff as in many situations confidentiality and integrity of data is more important than availability. Moreover, there could be other components in the system, beyond our control, which are susceptible to denial of service attacks, e.g., the Internet in a web-based application. The second tradeoff is one of security and performance. We have shown that performance degradation is modest (few percent) in most cases, and we feel that the gains in security outweigh the losses in performance.

## 6. RELATED WORK

A summary of existing techniques to protect the various components of a system is presented in Figure 5.

### 6.1 Application Software Design

Privilege separation [14][30][36] is a technique wherein the portions of a application that require high privileges to execute are extracted and executed in a separate program with high privileges. The rest of the application executes with lower privileges and communication between the two parts occurs via IPC calls. This approach works well when the security-sensitive resources are well-defined (e.g. ports < 1024, RSA private key). Privilege separation does not scale well, especially for complex applications like the browser. In the e-commerce transaction scenario, the parser, keyboard and mouse input and the screen output need to be protected. The corresponding modules account for around 60% of the browser's source.

Security-sensitive applications can be protected by executing them directly on a secure co-processor [50]. While isolation protects the application from vulnerabilities in other software, it does not address the issue of vulnerabilities in the application itself.

There are also many efforts addressing application specific vulnerabilities like interface spoofing in browsers. Tygar [46] discusses the use of interface personalization to thwart interface spoofing attacks. PwdHash [37] traps passwords and hashes them with some information from the communicating domain as salt before transmitting them. Other extensions like Trustbar [27] and SpoofGuard [16] aim to inform the user about the trustworthiness of the site based on information like SSL certificate, URL name and links in the content of the page. The security extensions reside in the same address-space as the browser and therefore are vulnerable to direct attacks such as buffer-overflow [11]. Techniques such as PwdHash are also complementary to our approach; we can implement password hashing in AppCores to limit the damage from password leaks, even by the remote communicating entity.

EROS Window System (EWS) [42] is an example of an essential system service (window manager) that is designed with software size and complexity in mind. EWS contains less than 4.5 KLOC and it provides robust traceability of user volition. Our GUI component includes user-input drivers and supports higher level widgets which accounts for the increased size. EWS illustrates that careful refinement of TCB components can further reduce the size and complexity of the TCB.

Static analysis and verification techniques can detect bugs and vulnerabilities [17][18][47]. However, they work best on relatively small code base due to scalability problems. Our work focuses on reducing the amount of security-sensitive code that needs to be analyzed or verified.

## 6.2 Operating System Kernel Design

There are many projects that aim to build smaller, more secure kernels, including security kernels [45], hypervisors [12], VMMs [22], microkernels [32], capability-based systems [41] and the Linux Security Modules (LSM) project [49]. Our goal of reducing software size is the same as that of microkernel research. AppCore research is focused on security-critical applications instead of operating system kernels. Moreover, the AppCore approach builds on system level TCBs, which can be VMM-based.

The XOM processor architecture [31] takes a different approach and provides security guarantees directly to the application, bypassing the operating system. The XOM approach eliminates the need for a trusted kernel or VMM at the lowest layer. Our work at the application and middleware layers is complementary to their approach.

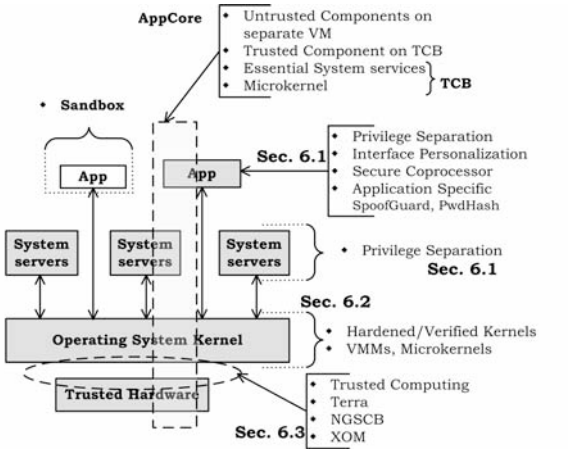


**Figure 5. Summary of Related Work. Shaded boxes represent security-sensitive components**

## 6.3 Trusted Computing Base

The Trusted Platform Module (TPM) defined by the TCG [10] provides a hardware specification for a TCB. TPM provide a secure environment with hardware support for authenticated booting, secure storage and secure IO. At a higher level, Integrity Measurement Architecture [39] provides integrity guarantees about the system runtime. Terra [22] and Microsoft's Next-Generation Secure Computing Base (NGSCB) [1] are similar initiatives that aim to provide a system-wide solution that includes hardware, operating system kernel and an execution environment.

Existing TCBs leave the application security issues to the applications. The AppCore recognizes that applications have grown in functionality and size, becoming too vulnerable. AppCores extend the good work on developing small operating system kernels and TCBs [15][28] to the application layer. A suitable underlying TCB is fundamental for guaranteeing the system wide security of AppCore.

The PERSEUS system architecture [35] is similar to the Nizza security architecture, in that both of them propose the execution of trusted components on a minimal execution environment. Our work focuses on developing trusted components for real-world applications, and demonstrating the viability of the Nizza architecture.

Gokyo [29] is a policy analysis tool that has been used to construct minimal TCBs for SELinux. The Gokyo tool analyzes existing policies with respect to integrity goals and identifies conflicts between policies and goals. Based on this information, administrators resolve conflicts manually and select components that form the TCB. Currently, we rely on manual analysis to identify components to form the TCB and AppCore. Automating parts of the process based on control and data flow analysis and policy analysis is the subject of future research.

## 7. CONCLUSION

The large size and high complexity of security-sensitive applications and systems software has hindered effective testing, resulting in end-user systems with a number of security vulnerabilities. We alleviate this problem by reducing the size and complexity of the security-sensitive application and systems software.

At the systems software level, we used the Nizza architecture as our TCB. Nizza relies on a kernelized TCB and on the reuse of legacy code using trusted wrappers to reduce the size of the TCB. Additionally, Nizza also allowed us to configure the components of the TCB depending on the needs of the security-sensitive application. Using Nizza, we were able to construct TCBs with around 100,000 lines of code. At the application level, we extracted security-sensitive portions of an already existing application into an AppCore. The AppCore was executed as a trusted process in the Nizza architecture and the rest of the application was executed as an untrusted process on an untrusted, virtualized, legacy operating system.

We implemented this approach for three real-world applications and found considerable reduction in code size and complexity (few tens of thousands of lines of code for AppCores compared to few hundred thousand lines of code for the current applications), with a modest loss in performance. In contrast to "monolithic" applications, the smaller sized AppCores make exhaustive testing or formal verification and validation possible and plausible.

## 8. ACKNOWLEDGEMENTS

We would like to thank Alexander Warg for his enormous contribution to the Mikro-SINA architecture and implementation. We would also like to thank our shepherd, Steven Hand, and the anonymous reviewers and for their valuable feedback. The authors from Georgia Tech were partially supported by NSF/CISE IIS and CNS divisions through grants CCR-0121643, IDM-0242397 and ITR-0219902, DARPA/IPTO through grant FA8750-05-1-0253, and Hewlett-Packard. The authors from TU Dresden were supported by grants from BMWi, DFG, and Intel.

## 9. REFERENCES

[1] Microsoft. Next-Generation Secure Computing Base. http://www.microsoft.com/resources/ngscb/default.mspx

[2] Mozilla Foundation. Mozilla Module Owners. http://www.mozilla.org/owners.html

[3] PeerSec Networks. MatrixSSL - Open Source Embedded SSL. http://www.matrixssl.org/

[4] Secunia. Vulnerability Report – Microsoft Internet Explorer 6. http://secunia.com/product/11/

[5] Secunia. Vulnerability Report – Mozilla Firefox 1.x. http://secunia.com/product/4227/

[6] Secunia. Vulnerability Report – X11 Windowing System (X11) 6.x. http://secunia.com/product/3913/

[7] Secunia. Vulnerability Report – Linux Kernel 2.4.x. http://secunia.com/product/763/

[8] Secunia. Check Point VPN-1 Products ISAKMP Buffer Overflow Vulnerability. http://secunia.com/advisories/11546/

[9] Snapgear. Snapgear Embedded Linux. http://www.snapgear.org

[10] Trusted Computing Group. *TCG Main Specification v1.1b*, https://www.trustedcomputinggroup.org/

[11] J. Bambenek, SANS Institute. BHO scanning tool and New Scam Targets Bank Customers. http://isc.sans.org/diary.php?date=2004-06-29.

[12] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A.Warfield. Xen and the Art of Virtualization. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, NY, Oct. 2003.

[13] V. Basili and D. Hutchens. An Empirical Study of a Complexity Family. In *IEEE Transactions on Software Engineering*, Volume 9, No. 6, November 1983, pp. 664-672.

[14] D. Brumley, D. X. Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proc. USENIX Security Symposium,* San Diego, USA. Aug 9-13, 2004.

[15] P. M. Chen and B.D. Noble. When Virtual is Better Than Real, In *Eighth Workshop on Hot Topics in Operating Systems,* May 2001, Elmau, Germany.

[16] N. Chou, R. Ledesma, Y. Teraguchi, D. Boneh and J. C. Mitchell, Client-side defense against web-based identity theft, In *11th Annual Network and Distributed System Security Symposium (NDSS '04)*, San Diego, February, 2004.

[17] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system specific programmer-written compiler extensions. In *4th USENIX OSDI*. San Diego, Oct. 2000.

[18] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *18th SOSP*. Banff, Canada, Oct. 2001.

[19] N. Feske, C. Helmuth: A Nitpicker's guide to a minimal-complexity secure GUI. In *Proc. of the 21st Annual Computer Security Applications Conference*, Tucson, Arizona, USA, Dec. 2005

[20] N. E. Fenton, N. Ohlsson., Quantitative Analysis of Faults and Failures in a Complex Software System. In *IEEE Trans. Software Eng*. 26(8): 797-814, 2000.

[21] Gaffney, J., Program Control Complexity and Productivity. In *Proceedings of the IEEE Workshop on Quantitative Software Models,* pg 179, October, 1979.

[22] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proc. of the 19th SOSP*, October 2003.

[23] H. Härtig. Security architectures revisited. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[24] H. Härtig, M. Hohmuth, N. Feske, C. Helmuth, A. Lackorzynski, F. Mehnert and M. Peter. The Nizza Secure-System Architecture. In *IEEE CollaborateCom 2005*. San Jose, USA. Dec 2005.

[25] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J.Wolter. The performance of µ-kernel-based systems. In *Proc. 16th ACM Symposium on Operating System Principles*, pp 66–77, Oct. 1997.

[26] C. Helmuth, A. Warg, and N. Feske. Mikro-SINA—Hands-on Experiences with the Nizza Security Architecture. In *Proceedings of the D.A.C.H Security 2005*, Darmstadt, Germany, March 2005.

[27] A. Herzberg and A. Gbara, TrustBar: Protecting (even Naïve) Web Users from Spoofing and Phishing Attacks, *Cryptology ePrint Archive*, Report 2004/155. 2004.

[28] Hohmuth, M., M. Peter, H. Härtig, and J. Shapiro. "Reducing TCB size by using untrusted components – small kernels versus virtual machine monitors", in *Proc. of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, 2004.

[29] T. Jaeger, R. Sailer, and X. Zhang, Analyzing Integrity Protection in the SELinux Example Policy, in *12th USENIX Security Symposium*, Washington D.C. USA, Aug. 2003.

[30] D. Kilpatrick, Privman: A Library for Partitioning Applications. In *USENIX Annual Technical Conference, FREENIX Track 2003*, pp 273-284. San Antonio USA, July 2003.

[31] D. Lie, C.A. Thekkath and M. Horowitz, Implementing an untrusted operating system on trusted hardware, In *19th ACM-SOSP,*2003, Bolton Landing, NY.

[32] J. Liedtke, On Micro-Kernel Construction, In *15th ACM Symposium on Operating System Principles*, Copper mountain Resort, Colorado, USA. Dec. 1995.

[33] T.J. McCabe, A Complexity Measure, *IEEE Transactions on Software Engineering*, SE-2 No. 4, pp. 308-320, Dec. 1976.

[34] X. Qie, R. Pang, L. L. Peterson, Defensive Programming: Using an Annotation Toolkit to Build DoS-Resistant Software. In *OSDI 2002,* Boston, Dec. 2002.

[35] B. Pfitzmann, J. Riordan, C. Stüble, M. Waidner and A. Weber. The PERSEUS System Architecture. *Research Report. IBM Research Division.* RZ 3335. Sept. 2001.

[36] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium*, Washington D.C, Aug. 2003.

[37] B. Ross, C. Jackson, N. Miyake, D. Boneh and J. C. Mitchell, Stronger Password Authentication Using Browser Extensions. In *14th Usenix Security Symposium*, Baltimore, USA, Aug. 2005.

[38] JH Saltzer and MD Schroeder, The Protection of Information in Computer Systems, *Proc. of the IEEE*, Vol.63, No.9, Sept. 1975, pp.1278-1308.

[39] R. Sailer, X. Zhang, T. Jaeger, and L. V. Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of Thirteenth USENIX Security Symposium*, pp 223--238, August 2004.

[40] B. Schneier. Software Complexity and Security. *Crypto-Gram Newsletter.* March 2000. http://www.schneier.com/crypto-gram-0003.html

[41] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A Fast Capability System. In *Proc. 17th ACM Symposium on Operating Systems Principles*. Charleston, SC, USA. Dec. 1999.

[42] J. S. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia, Design of the EROS Trusted Window System, In *Proc. USENIX Security Symposium*, San Diego CA, 2004

[43] V. Y. Shen, T. Yu, S. M. Thebaut, and L. R. Paulsen, Identifying Error-prone Software --- An Empirical Study, In *IEEE TOSE*, Vol. SE-11, pp. 317--323, April 1985.

[44] Shepperd, M., Ince, D.C., *Derivation and Validation of Software Metrics.* pp 37-40. Oxford Science Publications, 1993.

[45] R. Spencer, S.Smalley, P. Loscocco, M. Hibler, D.Andersen and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Procedings of the 8th USENIX Security Symposium*, Aug. 1999.

[46] J. D. Tygar and A. Whitten. WWW electronic commerce and Java Trojan horses. In *Proc. of the 2nd USENIX Workshop on Electronic Commerce*, Nov. 1996, pp. 243-250.

[47] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the ISOC Symposium on Network and Distributed System Security*, 2000.

[48] D. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/

[49] Wright, C., C. Cowan, S. Smalley, J. Morris, G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In the *Proceedings of the 2002 Usenix Security Symposium*, Aug 2002, San Francisco.

[50] B. Yee and D. Tygar. Secure coprocessors in electronic commerce applications. In *Proc. of the First USENIX Workshop on Electronic Commerce*, New York, July 1995.