

SecureBlue++: CPU Support for Secure Executables

Stephan Mielke

Institut für Programmierung und Reaktive
Systeme
Mühlenpförtstraße 23
Braunschweig, Deutschland
s.mielke@tu-bs.de

Nico Weichbrodt

Institut für Betriebssysteme und Rechnerverbund
Mühlenpförtstraße 23
Braunschweig, Deutschland
weichbr@ibr.cs.tu-bs.de

ABSTRACT

SecureBlue++ ist eine seit 2011 von IBM in Entwicklung befindliche CPU-Befehlssatzerweiterung zum verschlüsselten Verteilung, Speichern und Ausführen von Applikationen auf der POWER Architektur [2, 3, 9, 11]. Durch diese Abschottung ist es möglich den Angriffsvektor auf die eigene Software ohne große Änderungen zu reduzieren. Jedoch ergeben sich durch den Einsatz von SecureBlue++ nicht nur Vorteile, sondern auch Nachteile in Form von Leistungseinbußen und gravierenden Einschränkungen, welche eine Nutzung im gängigen Cloud-Computing-Szenario nahezu unmöglich machen.

1. MOTIVATION

Das Thema Cloud-Computing ist seit einigen Jahren immer mehr ein Thema und viele Unternehmen wollen nicht mehr eigene teure Rechenzentren betreiben, sondern nur noch Rechenzeit bei den sogenannten Cloud-Anbietern, wie zum Beispiel Amazon, Microsoft oder Google kaufen. Jedoch können diese Anbieter wegen dem amerikanischen Recht nicht garantieren, das ihre Server immer wie vom Kunden gewünscht die Anwendungen ausführen, oder ob sie zusätzlich die Daten der Anwendungen an Dritte weiter geben müssen [10]. Jedoch sind nicht nur staatliche Anordnungen eine Gefahr für die Anwendungen der Kunden, sondern ebenfalls infizierte Systeme oder nicht professionelle Administratoren im Cloud-Rechenzentrum.

Für diese Bedrohungen werden seit einiger Zeit Lösungen, wie zum Beispiel SecureBlue++, entwickelt [7]. Jedoch gibt es noch weitere Bedrohungen aber auf diese versucht SecureBlue++ keine Lösung zu finden. Diese wären:

- Programmierfehler und Sicherheitslücken in den Anwendungen
- Denial-of-Service Attacken in jeder Form

2. EINLEITUNG

In dieser Arbeit soll die Befehlssatzerweiterung SecureBlue++ aus dem Paper [3] vorgestellt werden. Hierfür wird zuerst erklärt was SecureBlue++ ist und im Anschluss die einzelnen Neuerungen erklärt. Hierfür werden in Abschnitt 3.4 die neuen Instruktionen und in Abschnitt 3.3 die neuen Register vorgestellt. System Calls sind für die gesicherte Ausführung einer Anwendung sehr problematisch, da die Anwendung den Kontrollfluss dem Kernel übergibt und hierbei, Geheimnisse der Anwendung verraten werden können. In Abschnitt 3.5 wird deswegen die Umsetzung mit System

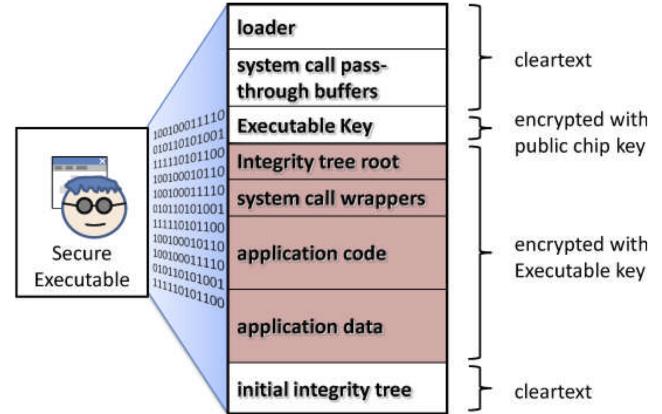


Abbildung 1: Aufbau von mit SecureBlue++ gesicherten ELF Binärdateien. [3]

Calls behandelt. Für moderne Anwendungen muss die SecureBlue++ Multi-threading unterstützen. Dies wird in Abschnitt 3.7 behandelt. Wie SecureBlue++ mit virtuellen Maschinen umgeht findet sich in Abschnitt 3.8. Im Anschluss werden Simulationen von SecureBlue++ getestet und aufgezeigt wie viel Leistungseinbußen auftreten können. Zum Abschluss folgt ein Vergleich von SecureBlue++ mit der Konkurrenztechnologie von Intel

3. SECUREBLUE++

IBM hat seit 2011 die SecureBlue++ CPU-Befehlssatzerweiterung in Entwicklung, welche auf der SecureBlue Technologie für Verschlüsselung mit Hardwareunterstützung basiert [1, 2]. Mittels dieser Erweiterung können Programme komplett isoliert von anderen Applikationen sowie dem privilegierten Betriebssystem ausgeführt werden. Hierfür wurden neue CPU-Instruktionen und mehrere neue Register eingeführt [3]. Des Weiteren ist der von der jeweiligen Anwendung belegten Arbeitsspeicher in die Bereiche „read-only“ und „execute“ und „no-execute“ disjunkt geteilt.

Ein weiteres Feature von SecureBlue++ ist die Möglichkeit eine Applikation verschlüsselt und geschützt durch einen Integritätsbaum auf der Festplatte vorzuhalten, um das Reverse Engineering oder das Manipulieren der Software zu verhindern. Hierfür besitzt jede CPU ein bei der Herstellung festgelegtes Public/Private Key-Paar, von dem der Public Key in einem „read-only“ Register zur Verfügung gestellt wird.

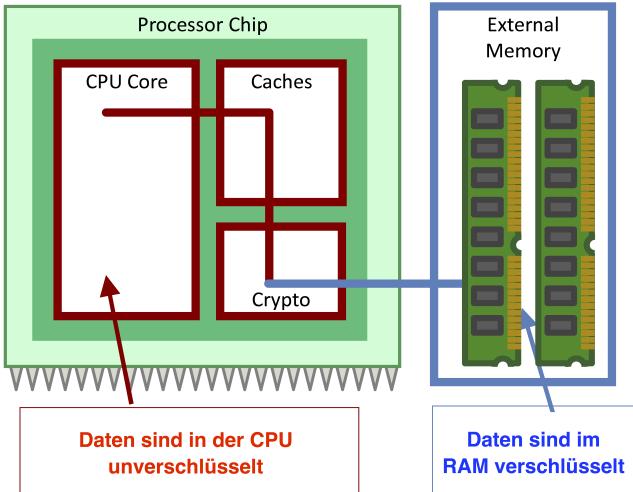


Abbildung 2: Grundidee: sensible Daten sind innerhalb der CPU unverschlüsselt und nur außerhalb verschlüsselt. [3]

3.1 Aufbau von gesicherten Anwendungen

Das Executable and Linking Format ELF Dateiformat ist ein im UNIX Umfeld weit verbreitetes Format für Anwendungen. SecureBlue++ ist konzipiert das es sich bei der Kompilierung von Anwendungen zu ELF Dateien in den beim Linken integriert. Hierdurch wird sichergestellt das keine Änderungen am Quellcode der Anwendung benötigt werden. In Abb. 1 wird der Aufbau einer mit SecureBlue++ gesicherten Anwendung aufgezeigt.

Der Loader besteht aus dem in Abb. 3b gezeigten unverschlüsselten Aufruf der `esm`-Instruktion (Abschnitt 3.4) mit dem öffentlichen Schlüssel verschlüsselte AES-Schlüssel der Anwendung sowie aus den in Abschnitt 3.6 gezeigten Metadaten. Des weiteren werden Buffer für die System Call Wrappers im unverschlüsselten Speicherbereich bereitgestellt.

Im Anschluss folgen mit dem AES-Schlüssel verschlüsselt die Daten der Anwendung wie dem Integrity Tree (Abschnitt 3.6), den System Call Wrappern (Abschnitt 3.5) und dem auszuführenden Code sowie den Daten der Anwendung. Am Ende befindet sich ein Hash-Wert zum validieren der ELF Datei.

3.2 Umsetzung per Hardware

Für Umsetzung von SecureBlue++ wurde hardwareseitig das Verhalten der POWER CPUs modifiziert.

RAM-Verschlüsselung.

In Abb. 2 wird die RAM-Verschlüsselung aufgezeigt, bei dem innerhalb der CPU die Daten unverschlüsselt sind und sobald diese die CPU verlassen verschlüsselt werden [11]. Dies wird völlig transparent für die Anwendung und nahezu transparent für das jeweilige Betriebssystem ausgeführt [3]. Für die Verschlüsselung kommt ein für die Anwendung eigens generierter AES-Schlüssel zum Einsatz.

Cache-Line-Verifikation.

Mittels des Integritätsbaums werden die Daten beim Laden in die CPU auf ihre Richtigkeit überprüft. Hierbei kann es allerdings zu Leistungseinbußen kommen, wenn die benötigen Teile des Baums nicht im Cache der CPU vorhan-

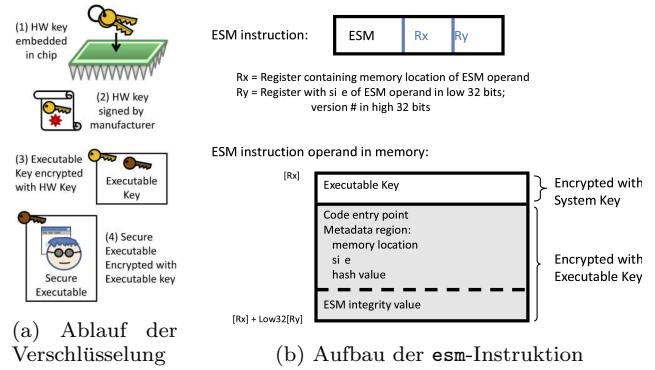


Abbildung 3: Verschlüsselung und `esm`-Instruktion [3]

den sind und deswegen ebenfalls geladen werden müssen. Beim Zurückschreiben von Cache Lines in den RAM wird der Baum dementsprechend automatisch aktualisiert [3]. Weitere Informationen folgen in Abschnitt 3.6.

Cache Line-Schutz.

Um das Auslesen von Cache Lines zu verhindern, wird jeder Cache Line, einer gesicherten Region, ein Label (*Memory Region ID MRID* welche mit der *Secure Executable ID SEID* verbunden ist), der dazugehörigen Anwendung, zugeordnet. Wenn eine Anwendung auf eine Cache Line zugreifen will, die mit einem Label versehen ist, werden die *MRID* Werte verglichen und bei Ungleichheit wird der Zugriff als ein Cache Miss behandelt [3].

Register-Schutz.

So lange sich die CPU im Secure-Modus befindet, ändert sie ihr normales Verhalten bei einem Interrupt. Bei jedem Interrupt außer dem (`sesc`) werden hardwareseitig alle Register mit der jeweiligen *SEID* in der *Thread Restore List (TRL)* gespeichert und anschließend mit Nullen überschrieben bevor der Kontrollfluss an den jeweiligen Interrupt Handler des Systems übergeben wird.

3.3 Neue Register

Zur Umsetzung von SecureBlue++ wurden mehrere neue Register eingeführt. Hierbei existieren, wie in Abb. 3a gezeigt, ein ROM¹ auf der CPU die Private/Public Keys beinhaltet. Der ROM für den Privaten Schlüssel ist nur innerhalb der `esm`-Instruktion von der CPU lesbar [3, 11].

Secure Executable ID Save/Restore (SEIDSR).

Das *SEIDSR*-Register kann nur mit privilegiertem Zugriff ausgelesen werden und gibt die aktuelle *SEID* zurück. Der Inhalt des Registers ist nur mit privilegierten Rechten beschreibbar und wird für die neu eingeführten Instruktionen benötigt.

Secure Executable ID (SEID).

Die CPU benutzt das *SEID*-Register (früher auch *EID*-Register genannt [2, 11]) für die interne Speicherung der aktuell ausgeführten durch SecureBlue++ geschützten Anwendung. Es ist nicht möglich den Wert dieses Registers

¹Wie genau der öffentliche Schlüssel gelesen werden kann, wurde nicht angegeben.

direkt zu manipulieren oder ihn auszulesen.

3.4 Neue Instruktionen

Es wurden vier neue Instruktionen als Interface eingeführt, damit Applikationen und das zugrunde liegende Betriebssystem die SecureBlue++ Erweiterung benutzen können.

Enable Secure Mode (esm).

Die **esm**-Instruktion ist in Abb. 3b aufgezeigt und bekommt eine Konfigurationsstruktur übergeben bei dem das Register Rx den Start und Ry die Größe der Struktur angeben. Die Konfiguration besitzt als erstes Attribut den **Executable Key**, welcher ein eigens für die Applikation erzeugter AES-Schlüssel ist und nur dieser wird, wie in Abb. 3a gezeigt, mit dem öffentlichen Schlüssel der CPU verschlüsselt. Alle weiteren Attribute sind mit dem **Executable Key** verschlüsselt. Bei dem **Code entry point** handelt es sich um einen Zeiger auf die Startfunktion der jeweiligen Applikation, zum Beispiel die **main** Funktion in einem C-Programm. Die **Metadata region** wird in Abschnitt 3.6 erläutert. Bei der **ESM integrity value** handelt es sich um einen Hash-Wert der Konfiguration um Manipulationen zu verhindern. Nachdem die Integrität der Konfiguration sichergestellt wurde, wird eine **SEID** generiert, in das **SEID**- und **SEIDSR**-Register geschrieben, die Eintragungen in den Verwaltungstabellen erzeugt und das Programm über den **Code entry point** gestartet.

restorecontext.

Das Betriebssystem muss beim Kontextwechsel zuerst die **SEID** aus dem **SEIDSR**-Register auslesen und sich diese zusätzlich in der Thread-Verwaltung speichern. Wenn das Betriebssystem eine mit SecureBlue++ geschützte Anwendung wieder aufnehmen, so muss das **SEIDSR**-Register mit entsprechenden und validen **SEID** belegt und die **restorecontext**-Instruktion aufgerufen werden. Im Anschluss werden automatisch durch die Hardware die Register und der Status aus der **TRL** wiederhergestellt und die Applikation nimmt ihre Arbeit wieder auf.

deletecontext.

Durch die **deletecontext**-Instruktion werden alle Ressourcen der durch das **SEIDSR**-Register bestimmten Anwendung, wie Metadaten, Register und Cache Lines sowie alle Eintragungen in den *Secure Executable Table* und *Protected Memory Table* gelöscht und können nicht mehr mit der **restorecontext**-Instruktion wiederhergestellt werden. Somit sollte diese Instruktion immer zum Beenden der jeweiligen Anwendung genutzt werden. Es entstehen keine Sicherheitslücken, wenn diese Instruktion durch Angreifer benutzt wird.

Secure Executable System Call (sesc).

Diese Instruktion wird zwingend für die Umsetzung von System Calls benötigt, weil im Secure-Modus das Verhalten der eigentlichen **sc**-Instruktion, wie in Abb. 4 dargestellt, verändert wurde. Weitere Informationen zur Funktionsweise der **sesc**-Instruktion folgt in Abschnitt 3.5.

3.5 System Calls

System Calls dienen als Interface für Anwendung mit dem jeweiligen Betriebssystem. Jedoch ist nach dem Bedrohungsszenario aus Abschnitt 1 das Betriebssystem mit den dazu-

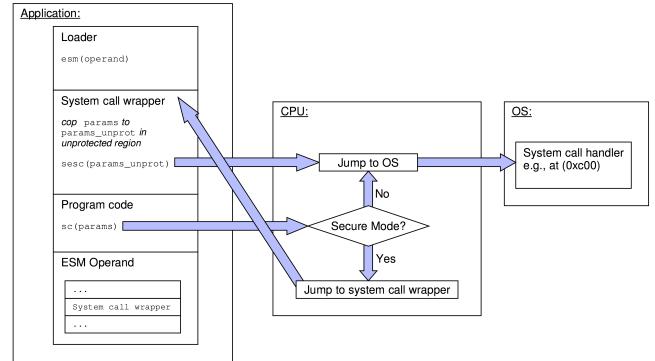


Abbildung 4: Grundidee: Die **sc**-Instruktion wird im Secure-Modus von einem eigenen Handler abgefangen und verarbeitet. [3]

gehörigen Interrupt Handlern als nicht vertrauenswürdig zu erachten und muss aus diesem Grund aus der *trusted computing base TCB* ausgeschlossen werden. Des Weiteren müssen die Rückgabewerte von System Calls auf ihre Sinnhaftigkeit überprüft werden, um gegen Iago Attacken abgesichert zu sein [4].

Bei jedem Aufruf der **sc**-Instruktion wird von der CPU überprüft, ob sich die Applikation im Secure-Modus befindet oder nicht. Ist der Secure-Modus nicht aktiviert wird der System Call Handler des Betriebssystems aufgerufen und die Abarbeitung des System Calls erfolgt wie gewohnt durch einen System Call Interrupt. Ist die Anwendung jedoch im Secure-Modus wird ein eigener in der Applikation vorhandener System Call Handler aufgerufen. Dieser kopiert die Daten aus der dem geschützten Arbeitsspeicher in ungeschützte Regionen und führt im Anschluss mit dem **sesc**-Interrupt den System Call aus. Nach der Beendigung des System Calls werden die Ergebnisse auf Sinnhaftigkeit überprüft und in den gesicherten Arbeitsspeicher überführt.

Aus Performance-Gründen wird beim **sesc**-Interrupt der Register-Schutz ausgeschaltet, weswegen der System Call Wrapper die Register für die Übergabe von Werten verwenden kann aber sich selbst für die Sicherheit der Daten kümmern muss, um nicht versehentlich Geheimnisse zu verraten [3]. Durch diese Umsetzung von System Calls müssen keine Änderungen am Quellcode der Anwendung oder der zugrunde liegenden Bibliotheken, wie der **libc**, vorgenommen werden. Jedoch können nicht alle System Calls mit dieser Art Wrapper durchgeführt werden, da manche entweder den geschützten Speicher der Anwendung manipulieren oder den Status der Register verändern.

signal.

Das Betriebssystem hat bei durch SecureBlue++ geschützten Applikationen keine Möglichkeiten das **Program Counter**-Register zu verändern. Somit ist es nicht mehr möglich, dass das Betriebssystem wie üblich den PC auf den gewünschten Signal Handler zu verschiebt. Um den **signal** System Call trotzdem zu unterstützen, wird beim Aufruf der **esm**-Instruktion ein übergeordneter Handler registriert, der auf Anwendungsebene die jeweils gewünschte Aktion ausführt [3]. Damit kann das Betriebssystem höchstens eines der angebotenen Aktionen von der Anwendung ausführen lassen und keine anderen für das Programm möglicherweise gefährliche

chen Aktionen. Die Umsetzung des Signal Handlers ist für das Betriebssystem völlig transparent.

fork.

Der **fork** System Call kann nicht über das Betriebssystem ausgeführt werden, da das Betriebssystem keinen lesenden Zugriff auf die jeweiligen Speicherbereiche besitzt. Deswegen wird durch den Wrapper einen Signal an die Applikation gesendet, welches die Anwendung veranlasst seinen benutzen Speicher selbst zu duplizieren und diesem im Anschluss einem neuen mit SecureBlue++ gesichertem Prozess zu zuweisen.

clone.

Um Multi-threading zu ermöglichen ist eine Unterstützung des **clone** System Calls zwingend erforderlich. Wie auch **fork** benötigt **clone** eine Unterstützung durch die gesicherte Anwendung. Weitere Informationen hierzu folgen in Abschnitt 3.7.

exit.

Damit keine Geheimnisse aus einer beendeten Anwendung entweichen können, wird das Programm nach dem Aufruf von **exit** in eine Endlosschleife aus **nop**-Instruktionen versetzt. Hierdurch wird verhindert das nach der Beendigung der Anwendung irgendwelche potenziell gefährlichen Aktionen ausgeführt werden.

3.6 Umsetzung über Software

Neben dem System Call Wrapper gibt es weitere durch Software umgesetzte Funktionalitäten mit deren Hilfe SecureBlue++ eine gesicherte Ausführung von Anwendungen ermöglicht. Diese Bestandteile werden in die zu sichernde Anwendung statisch hinein kompiliert.

Integrity Tree.

Der Integrity Tree dient neben der Verschlüsselung zum Schutz des Arbeitsspeichers vor Manipulationen. Der Baum ist nach dem Key/Value-Prinzip aufgebaut, bei dem jede zu schützende Cache Line auf einen Hashwert in den Blättern abgebildet wird. Die Knoten des Baumes schützen immer ihre Kindknoten vor Manipulationen durch das Abbilden von Kindknoten auf Hash-Werte. Um den Baum möglich flach zu halten, kann jeder Knoten acht Kindknoten beherbergen und schützen. Cache Lines, die im Cache verändert wurden, werden Markiert und erst wenn diese aus dem Cache entfernt werden sollen in den Baum zurück geschrieben. Hierbei muss jedoch vom Blatt an alle Elternknoten ebenfalls mit aktualisiert werden, da die Elternknoten die Hash-Werte der Kindknoten mit überwachen.

Metadata.

Die Metadaten einer gesicherten Anwendung werden durch den Integrity Tree geschützt, bei jedem wieder Aufruf der Anwendung neu in den Cache geladen sowie validiert und werden in der so genannten *Metadaten Region* gespeichert. Der Zustand wird, wie in Abb. 3b aufgezeigt, beim Aufzug der **esm**-Instruktion übergeben. Die Metadaten bestehen unter anderem aus den Signal Handlern, die bei der **signal**-Instruktion aufgerufen werden, die hinein kompilierten System Call Wrapper und der *die Memory Region Mapping Table (MRMT)*. Des weiteren beinhalten die Metadaten die

Adresse der *TRL*.

Secure Executable Table.

Bei der *Secure Executable Table (SET)* handelt es sich um eine nur in der CPU² befindliche Datenstruktur, welche alle aktiven gesicherten Anwendungen beinhaltet und nur durch die CPU mit den neu eingeführten Instruktionen geändert werden kann. Die Größe der *SET* ist hardwareseitig limitiert³ und hierdurch ist ebenfalls die maximale Anzahl von gleichzeitig zu benutzenden durch SecureBlue++ gesicherten Anwendungen limitiert. Jeder Eintrag besteht aus den diesen Eigenschaften:

- Secure Executable ID (*SEID*)
- Hash-Wert der Metadaten zur Validierung
- Zeiger auf Metadaten Region im Speicher

Memory Region Mapping Table.

Die *MRMT* ermöglicht die gesicherte Anwendung eine bei Cache Misses für schnelle Zuordnung von *MRIDs* auf die jeweiligen Speicherbereiche mit dem dazugehörigem AES-Schlüssel. Alle Informationen dieser Tabelle können ebenfalls aus der *Protected Memory Table (PMT)* ausgelesen werden, ermöglicht die *MRMT* das Teilen von Speicherbereichen für das Multi-threading. Die Tabelle ist eine „read-only“ Datenstruktur und jeder Eintrag besitzt die folgenden Attribute:

- Angaben zum Start und Größe der Region im Speicher.
- Der jeweilige AES-Schlüssel für die Region, da jede Region bei geteilter Speicher von mehreren Anwendungen benutzt werden könnte.
- Die Memory Region ID (*MRID*) zur Bestimmung der jeweiligen Region in der *PMT*.

Protected Memory Table.

Diese Tabelle ist nur indirekt über die neuen Instruktionen manipulierbar und hält alle Verwaltungsinformationen zu den einzelnen gesicherten Speicherbereichen aller gesicherten Anwendungen in der CPU² vor. Grundsätzlich hat jede Anwendung Zugriff auf die verschlüsselten Speicherbereiche, so lange der jeweilige AES-Schlüssel bekannt ist. Anwendungen mit einer passenden *SEID* in *SEID1* oder *SEID2* haben lesenden Zugriff auf die jeweilige Zeile und können somit den AES-Schlüssel abfragen. Jeder dieser Zeilen besteht unter anderem aus den folgenden Daten:

- Memory Region ID (*MRID*)
- *SEID* des Erstellers als *SEID1*
- Eine weitere *SEID* als *SEID2*
- Statuswert um die Region auf „read-only“ zu stellen
- Größe und Start der Region
- AES-Schlüssel zur Region

²Die genaue Position wurde bisher nicht genau bekannt gegeben.

³Die genaue Größe wurde bisher nicht spezifiziert.

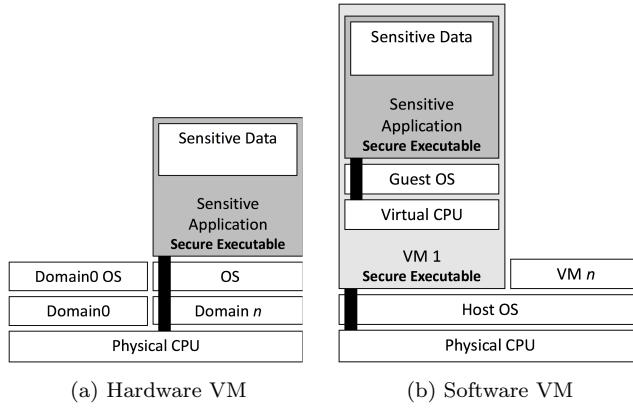


Abbildung 5: Arten von VMs [3]

Thread Restore List.

In der *TRL* werden hardwareseitig die Registerzustände im bei einem Interrupt gespeichert und durch den Aufruf der **restorecontext**-Instruktion wiederhergestellt. Nach dem Wiederherstellen eines Kontextes wird der Eintrag als benutzt markiert und kann nicht erneut wiederhergestellt werden.

3.7 Multi-threading

Multi-threading ist in der heutigen Zeit nicht mehr weg zu denken und muss deshalb für geschützte Anwendungen zur Verfügung stehen. Beim Multi-threading bestehen zwei verschiedene Varianten die jeweils unterschiedliche Vor- und Nachteile besitzen und durch SecureBlue++ anders behandelt werden müssen.

User Threads.

User Level Threads werden auf Benutzerebene implementiert und durch einen eigenen Dispatcher verwaltet und ausgeführt. Aus diesem Grund kann der Kernel Scheduler nicht die einzelnen User Level Threads verwalten, weswegen keine echte Nebenläufigkeit unterstützt wird. Da der Dispatcher selbst von der Anwendung stammt, kann ein angepasster Scheduler mit in die Anwendung kompiliert werden. Hierdurch müssen keine weiteren Änderungen am Programm, für die Unterstützung von Multi-threading, vorgenommen werden.

Kernel Threads.

Diese Art von Threads werden durch den Kernel verwaltet und ausgeführt. Hierfür wird eine Unterstützung durch die CPU benötigt, da für eine gesicherte Anwendung mehrere Kontexte durch die unterschiedlichen Threads verwaltet werden müssen. Normalerweise werden Threads mit dem `clone` System Call erstellt. Hierfür stellt SecureBlue++ einen POSIX kompatiblen Wrapper zur Verfügung, weil die Betriebssystem eigene Implementierung keinen Zugriff die *TRL* besitzt. Beim Aufruf von `clone` wird ein neuer Eintrag in der *TRL* mit den entsprechenden Statuswerten angelegt. Des weiteren ist es dem Kernel Scheduler möglich zwischen den verschiedenen noch nicht gestarteten in der *TRL* gespeicherten Kontexten auszuwählen.

3.8 Virtuelle Maschinen

SecureBlue++ wurde designt um mit Soft- und Hardware

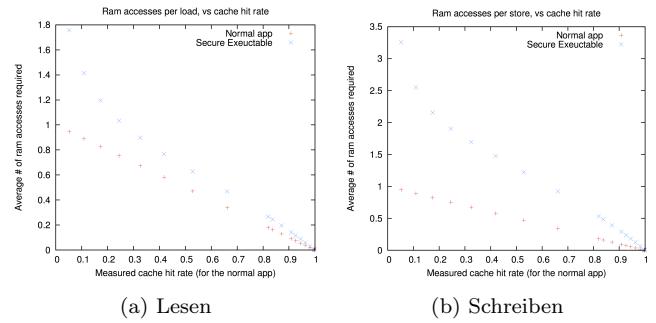


Abbildung 6: Anzahl der durchschnittlich benötigten RAM-Zugriffe beim lesen / schreiben von Cache Lines[3]

basierten virtuellen Maschinen, die in Abb. 5 aufgezeigt sind, zusammenzuarbeiten.

Hardware VMs.

bei Hardware VMs oder auch „bare-metal VMs“ genannt, handelt es sich um VMs ohne einen Softwarehypervisor, welcher die privilegierten Instruktionen des Gastsystems (in Abb. 5a Domain n) abfängt. Somit verhält sich eine Hardware VM wie ein normales System und kann genauso Applikationen die mit SecureBlue++ gesichert sind ausführen. Für IBM ist diese Art der VM die eigentliche Zielgruppe, da diese am weit verbreitetsten im high-performance Sektor sind. Wie bei normalen System muss nur die jeweilige Applikation innerhalb der TCB liegen und das Betriebssystem der VM muss nicht vertraut werden. Das Domain 0 System startet bzw. stellt die gesicherte Anwendung wieder her indem es die `restorecontext`-Instruktion mit der jeweiligen *SEID* aufruft. Um an die *SEID* zu kommen, muss das Domain 0 System bei einem Hypervisorinterrupt das *SEID*-Register auslesen.

Software VMs.

Im Gegensatz zu Hardware VMs besitzen Software VMs einen Softwarehypervisor, der die privilegierten Instruktionen des Gastsystems abfängt und an das gastgebende System weiterleitet und ausführen lässt. Dieser Umstand zwingt einen anderen Ansatz als bei Hardware VMs, da fast alle neuen Instruktionen aus Abschnitt 3.4 privilegiert sind und deswegen der Softwarehypervisor Zugriff auf die geschützte Anwendung benötigt. Aus diesem Grund muss das Gastsystem, wie in Abb. 5b gezeigt, in eine gesicherte Anwendung überführt werden. Somit muss jedes System das Instruktionen emuliert in die *TCB* mit aufgenommen werden, jedoch muss das gastgebende System nicht in die *TCB* aufgenommen werden. Dennoch kann die Applikation vom Gastsystem geschützt werden, in dem die zu sichernde Anwendung in eine virtuelle Secure Executable überführt wird. Hier durch wird das Gastsystem vor dem gastgebenden System geschützt sowie die Anwendung vor beiden Systemen.

4. BENCHMARK

Das SecureBlue++ Projekt ist seit 2011 in Entwicklung und es gibt aktuell keine echten POWER CPUs die die SecureBlue++ Erweiterung unterstützen. Deswegen sind alle Benchmarks und Performance-Analysen sehr mit Vorsicht zu genießen und die echten Leistungsdaten können sehr viel

	SB	SGX [8]
Feinheit des Schutzes	Prozess	Virtuelle Speicherregion
OS in der TCB	N	N
Limitiert die Anwendung	N	N
RAM Validierung hardwarebeschleunigt	N	J
Dynamisch wachsender Speicher	J	N
Anwendung ist Verschlüsselt	J	N
Geheimnisse nur im speziellen Bereichen	N	J
An die CPU gebunden	J	N

Tabelle 1: Vergleich zwischen SecureBlue++ (SB) und SGX, Grün ist ein Vorteil und Rot eine Limitierung [7]

schlechter als die simulierten sein.

Für die Benchmark-Simulation wurde eine Anwendung emuliert die auf eine Sequenz von 10^6 Adressen in einer Exponentialverteilung zugreift. Für den Cache wurde eine Größe von 4096 Zeilen mit 8 Spalten und jeweils einer Größe von 32 Wörtern gewählt. Des weiteren wurden die Ver- und Entschlüsselungen nicht simuliert. Diese würden die Ergebnisse der sicheren Anwendung nochmals in der Reaktionszeit verschlechtern. Hauptsächlich soll mit dem Benchmark gezeigt werden, wie der Integrity Tree aus Abschnitt 3.6 die Anzahl der benötigten RAM-Zugriffe für schreibenden bzw. lesenden Zugriff auf die Cache Lines beeinflusst.

Für eine normale Anwendung ist das Lesen von Cache Lines in Abb. 6a eine Gerade, weil bei jedem Cache Miss nur die jeweilige Line mit einem einzigen RAM-Zugriff geladen werden muss. Bei der simulierten gesicherten Anwendung muss für jede zu landende Cache Line auch der jeweilige Knoten des Integrity Tree mit all den Elternknoten geladen werden, da die Cache Lines mit dem Integrity Tree validiert werden.

Beim Beschreiben von Cache Lines in Abb. 6b muss bei einer normalen Anwendung nur die jeweilige Line geändert werden. Sollte diese noch nicht im Cache vorliegen, muss diese mit einem einzigen Zugriff geladen werden. Bei der gesicherten Anwendung muss hierfür ebenfalls wie beim Lesen der jeweilige Knoten des Integrity Trees geladen werden. Des weiteren muss der Baum geändert werden und diese Änderungen werden bis zur Wurzel des Baums durch gedrückt. Deswegen steigen die Anzahl der RAM-Zugriffe sehr viel stärker als beim Lesen an.

5. VERGLEICH MIT INTEL SGX

Neben SecureBlue++ gibt es auch weitere Ansätze zur isolierten Ausführung von Anwendungen. Am bekanntesten ist Intel SGX [5], welche mit SecureBlue++ einige Gemeinsamkeiten besitzt. Die größten Unterschiede zwischen beiden Varianten ist, das SecureBlue++ nativ die Verschlüsselung der Anwendung unterstützt und das die gesamte Anwendung in der sicheren Enklave läuft. Bei SGX hingegen kann eine Anwendung aus mehreren Enklaven bestehen und muss nicht für jede CPU einen neuen Schlüssel erhalten.

Tabelle 1. vergleicht SGX mit SecureBlue++. Hierbei bedeutet Grün das es ein Vorteil und Rot ein Nachteil ist. Bei Gelb ist es nicht eindeutig ob es sich um einen Vorteil oder Nachteil handelt.

6. ZUSAMMENFASSUNG

IBMs SecureBlue++ ist in der Theorie eine interessante Befehlssatzerweiterung für die gesicherte und isolierte Ausführung von Anwendungen, die mit den Konkurrenzprodukten anderer Firmen wie Intels SGX mithalten. Leider existiert SecureBlue++ noch nicht als benutz- und evaluierbare Version auf einer realen CPU. Somit sind alle Überlegungen und Vorteile nur theoretischer Natur. Des weiteren existieren versteckte Einschränkungen die den Einsatz in Coud-Computing-Bereich sehr erschweren.

Der AES-Schlüssel für die Applikation ist mit dem öffentlichen Schlüssel von einer CPU verschlüsselt. Somit kann die Anwendung auch nur auf dieser einen CPU ausgeführt werden. Jedoch hat man im Coud-Computing-Bereich meist keinen direkten Einfluss auf welcher physischen Maschine die Anwendung ausgeführt wird.

Viele wichtige Informationen wie zum Beispiel die Limitierung der *SET* oder auf welchen Komponenten die *PMT* oder *MRMT* gespeichert werden fehlen aktuell.

Durch die Verschlüsselung der Anwendung wäre es theoretisch möglich auf einem kompromittierten System eine gesicherte Anwendung als Malware auszuführen ohne das der Betreiber des Rechenzentrums analysieren kann was diese Anwendung macht [6].

In den kommenden POWER9 CPUs sollen erste Versionen von SecureBlue++ zur Verfügung stehen und dann wird sich zeigen, ob SecureBlue++ die aufgestellten Erwartungen erfüllen kann.

7. Literatur

- [1] Ibm extends enhanced data security to consumer electronics products. *IBM Extends Enhanced Data Security to Consumer Electronics Products*, Apr 2006.
- [2] R. Boivie. Secureblue++: Cpu support for secure execution. Technical report.
- [3] R. Boivie and P. Williams. Secureblue++: Cpu support for secure executables. Technical report.
- [4] S. Checkoway and H. Shacham. *Iago attacks: Why the system call api is a bad untrusted rpc interface*, volume 41. ACM, 2013.
- [5] V. Costan and S. Devadas. Intel sgx explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 2016. <https://eprint.iacr.org/2016/086>.
- [6] S. Davenport and R. Ford. Sgx: the good, the bad and the downright ugly. *Virus Bulletin*, 2014.
- [7] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley. Iso-x: A flexible architecture for hardware-managed isolated execution. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 190–202. IEEE, 2014.
- [8] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.
- [9] D. Pendarakis. Hardware support for malware defense and end-to-end trust, 10 2012.
- [10] R. Poplak. Edward snowden prism and the privacy we never had. *Mail & Guardian*, 12, 2013.
- [11] D. P. Richard Harold Boivie. Protecting application programs from malicious software or malware, 09 2011.