Technische Universität Berlin

Forschungsberichte der Fakultät IV
Elektrotechnik und Informatik

Enhanced BARM — Authentic Reporting to External Platforms

Patrick Stewin

Technische Universität Berlin

Technical Report

Bericht-Nummer: 2014-03
ISSN: 1436-9915

September 2014

Enhanced BARM[*] — Authentic Reporting to External Platforms[†]

Patrick Stewin
Security in Telecommunications, TU Berlin
patrickx@sec.t-labs.tu-berlin.de

1 Introduction

Our motivation for implementing an authentic channel application for state reporting is to deliver BARM's measurement results to an external platform protected from *Direct Memory Access* based malicious software (DMA malware). The external communication partner can evaluate the transmitted measurements to check if the counterpart has been attacked by DMA malware. The measurement results are based on processor register values (see [1, Section 3]). To exclude malware on the *Network Interface Card* (NIC) from modifying and forging outgoing network packets we need a secure communication channel. Such a channel not only assures confidentiality, integrity, and freshness of the transmitted data, but also authenticity of the channel endpoints. To implement such a channel we adapt the concept of a trusted channel that we presented in prior work [2, 3].

A trusted channel is a communication channel that implements secure channel properties and additionally binds communication endpoint state information to the communication session. Deploying a secure channel based on IPsec or TLS is insufficient in our case. IPsec or TLS based secure channels ensure confidentiality, integrity and freshness of the transmitted data. However, these channels are not bound to the actual communication endpoint. We implement the trusted channel based reporting application for BARM to prevent at least the following attacks. Such attacks could be conducted by malware that is executed on the network interface card. The malware could prevent BARM from communicating with the external platform by blocking or corrupting outgoing network packets. An attacker could also use such malware to steal key material, which is present in the host main memory, of the secure channel via DMA. Afterwards, the attacker can conduct a MitM attack. The malware could also relay the platform state information of a third platform, which is not attacked by DMA malware, to the external administrator platform. This means that the administrator platform could be tricked by conducting a *relay attack*.

We require at least secure channel properties (requirement R1) to ensure confidentiality, integrity, and freshness of the transfered data for our authen-

[*] The Bus Agent Runtime Monitor BARM is presented in [1].

[†] This technical report is based on Chapter 6 of the author's PhD thesis *Detecting Peripheral-based Attacks on the Host Memory.*

tic reporting channel [see 2, p.32]. The confidentiality property ensures that the attacker only gets a minimal amount of information. The integrity property ensures that corrupted network packets will be revealed immediately. The freshness property prevents the attacker from conducting a replay attack where a valid communication session is recorded to be replayed at some later time. To reveal an attack that is blocking packets that contain platform state information we introduce so-called *heartbeat* messages as payload that has to be sent during the communication session. A heartbeat in computing is a signal that indicates that, e. g., the corresponding software is still up and running [4].

The heartbeat message consists of the current BARM measurement and log information if an attack was prevented. If the network interface card has been stopped due to an attack heartbeat messages will no longer be received by the external platform. This behavior is interpreted by the external platform as a NIC-based attack. The transmitted information also includes state changes. State changes were also considered by the trusted channel concept [2, 3], but efficient and effective runtime monitoring with negligible performance overhead as implemented in BARM was missing [see 2, p.36]: "A state change on one platform is noticed by CM (an efficient monitoring agent assumed [. . .]". BARM represents the missing "monitoring agent" in our DMA malware scenario.

Compared to prior work [2, 3] the trust and adversary model for our DMA malware scenario does not require trusted computed mechanisms as proposed by the Trusted Computing Group (TCG)[1]. Our channel is not based on a *Trusted Platform Module* (TPM, see [5] for example) since we do not rely on load-time code integrity checks. Channel linkage to load-time measurements stored in a TPM is not required in our application. We require that the results determined by BARM are bound to our channel (requirement R2). This is necessary during the negotiation of the communication session as well as during the communication session itself.

Please note that we do not count on the *Input/Output Memory Management Unit* (I/OMMU) such as *Intel's Virtualization Technology for Directed I/O* (Intel VT-d, see [6]) implementation. This is another difference to the trust model of our prior work [2]. This technology was introduced shortly before our results were published [2]. This means that previous authors had not been confronted with I/OMMU issues as presented in [7, Section 8]. Previous works assumed that drivers capable of configuring the I/OMMU correctly exist. For this work we analyzed the I/OMMU in more detail and we decided not to rely on VT-d for our authentic reporting channel. Our prior work also introduced the requirement for privacy (requirement R3). This means, the channel considered the least information paradigm to minimize the disclosure of platform state information to only the bare necessities.

The main contributions of this work are as follows:

- **Authentic reporting channel that excludes the network interface card from the endpoint:** Malware executed on the network interface card is able to steal secret key material from the main memory to conduct a MitM attack. Hence, we developed an authentic reporting channel that ensures that only the host CPU is the communication endpoint. Our channel is based on the secure channel protocol TLS. We adapt the TLS protocol to exchange BARM measurements and to bind the channel to its

[1] `http://www.trustedcomputinggroup.org/`

supposed endpoint. An additional feature of our communication channel is platform state change reporting. This means that our runtime monitor BARM permanently delivers every state change regarding DMA malware to the communication partner via the authentic reporting channel. Our TLS modifications are based on TLS extensions. This means that our channel is compliant with the TLS specification. Our TLS compliant channel is the first channel that considers platform state reporting regarding DMA malware. It is also the first channel that is based on an implemented effective and efficient runtime monitor to report state changes. Previous work only assumed the presence of such a runtime monitor.

- **Analysis of the ethernet controller:** Our communication channel requires the network interface card. Hence, the ethernet controller will induce bus transactions. These bus transactions must be considered by BARM. This work demonstrates how the ethernet controller can be integrated into BARM's detection model, i. e., how to utilize the ethernet controller as an additional bus agent.

- **Enhancing BARM's detection model with a new parameter:** The ethernet controller transfers data packets, which size is greater than the size of address pointers and keystroke codes. We demonstrate that the cache line size is an important parameter for BARM's detection model. The cache line size is necessary to compute the number of expected bus transactions correctly.

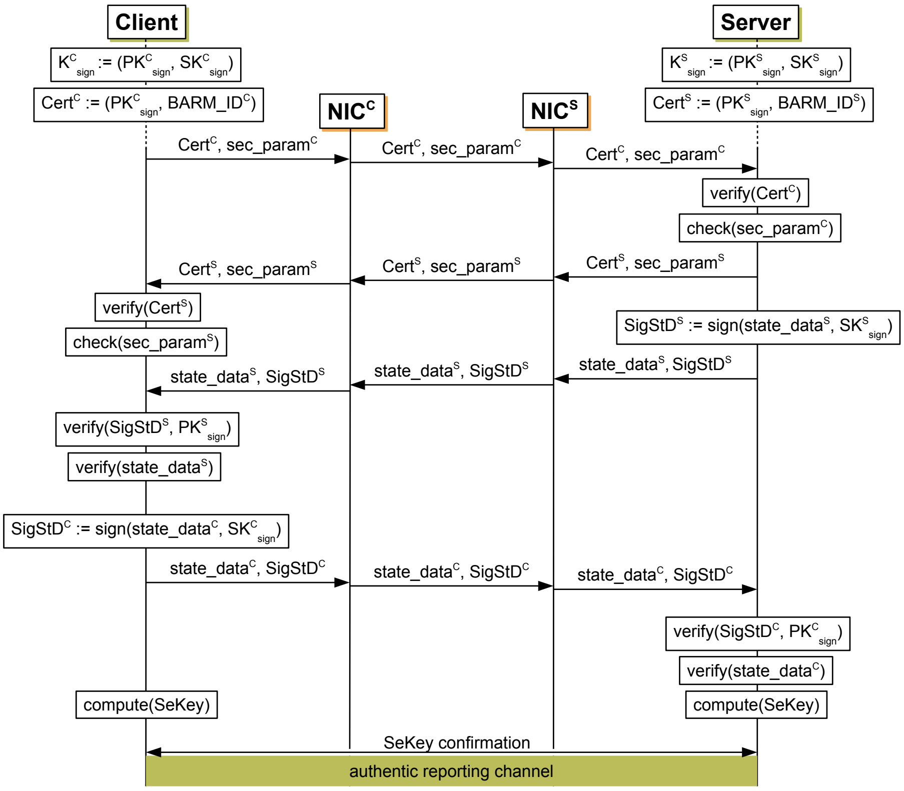
- **Exploiting additional performance monitoring unit events:** We demonstrate that certain performance monitoring unit configurations can be exploited to distinguish between memory read bus transactions and memory write bus transactions. This enables us to check if the number of expected read bus transactions and expected write bus transactions that are caused by the ethernet controller are correctly determined by BARM's detection model.

The following section starts with a description of the authentic reporting channel model. Afterwards, we explain how we implemented this model.

2 Implementation Independent Model

Our channel model considers client C (target platform) and server S (external platform) communication. Each endpoint may request platform state information (i. e., BARM measurements) of the peer. A local security policy determines what exactly happens after the platform state information of the peer has been evaluated. Our authentic reporting channel is controlled by host CPU software. The channel can be negotiated through a potentially compromised network interface card. We describe a high-level protocol for negotiating and maintaining an authentic reporting channel in the following section. Please note, in the following we omit the superscript C and S due to the symmetric protocol characteristic.

Figure 1: Negotiating an Authentic Reporting Channel


Client
Server
K^C_{sign} := (PK^C_{sign}, SK^C_{sign})
$Cert^C$:= (PK^C_{sign}, $BARM\_ID^C$)
NIC^C
NIC^S
K^S_{sign} := (PK^S_{sign}, SK^S_{sign})
$Cert^S$:= (PK^S_{sign}, $BARM\_ID^S$)
$Cert^C$, sec_param^C
$Cert^C$, sec_param^C
$Cert^C$, sec_param^C
verify($Cert^C$)
check(sec_param^C)
$Cert^S$, sec_param^S
$Cert^S$, sec_param^S
$Cert^S$, sec_param^S
verify($Cert^S$)
check(sec_param^S)
$SigStD^S$:= sign($state\_data^S$, SK^S_{sign})
$state\_data^S$, $SigStD^S$
$state\_data^S$, $SigStD^S$
$state\_data^S$, $SigStD^S$
verify($SigStD^S$, PK^S_{sign})
verify($state\_data^S$)
$SigStD^C$:= sign($state\_data^C$, SK^C_{sign})
$state\_data^C$, $SigStD^C$
$state\_data^C$, $SigStD^C$
$state\_data^C$, $SigStD^C$
verify($SigStD^C$, PK^C_{sign})
verify($state\_data^C$)
compute(SeKey)
compute(SeKey)
SeKey confirmation
authentic reporting channel

Negotiating an authentic reporting channel between Client C (target platform) and Server S (e. g., external administrator platform).

NIC	**Network Interface Card**
K_{sign}	**Asymmetric signing key pair (PK_{sign}, SK_{sign}) bound to host CPU software components**
PK	**Public key**
SK	**Secret key**
$Cert$	**Certified public key part of key pair bound to host CPU software components**
$BARM\_ID$	**Host CPU software components identifier**
sec_param	**Required security parameters**
$state\_data$	**Platform state data determined by BARM**
$SigStD$	**Signature of platform state data**
$SeKey$	**Session key**

2.1 Negotiating an Authentic Reporting Channel

One important idea of our authentic reporting channel is to prevent platform peripherals from accessing sensitive information that is related to the channel such as secret key material. Only host CPU software is allowed to use sensitive channel information. Please note, a peripheral could steal such information via DMA. However, BARM will reveal and stop this kind of DMA attack, see [1, Section 5.3]. Figure 1 depicts the handshake protocol for negotiating an authentic reporting channel for BARM. In order to conduct the handshake, both parties require a signing key K_{sign} that is an asymmetric key pair, i. e., $K_{sign} := (PK_{sign}, SK_{sign})$. Furthermore, both peers require a certificate $Cert$, which includes PK_{sign} as well as a host CPU software components identifier ($BARM\_ID$). This certificate is issued by a trusted party, which can be the external administrator platform. The signing key and the certificate are created before negotiating an authentic reporting channel. Each peer verifies the certificate including $BARM\_ID$ of its counterpart.

The creation of the channel begins with the negotiation of security parameters. This means that each party sends its certificate as well as security requirements in the form of security parameters to the peer. The security parameters determine which party reports its platform state information. Each peer checks if the security requirements of the counterpart are acceptable. In the next step, each party sends its platform state data (the current BARM measurement) to the peer. The state data is digitally signed and the corresponding signature is transmitted together with the state data. This ensures that the received state data has been sent by the expected communication partner. Both parties verify the signature with PK_{sign} that was sent by the peer as part of the certificate $Cert$. If the signature is valid both parties verify the state data. The handshake may be aborted due to DMA malware that attacks the peer. This is the case when the transmitted BARM measurement result is greater than the tolerance value $\mathcal{T}$ (see [1, Section 4.3]). After both client and server have verified the exchanged data successfully the same session key is computed and confirmed by both platforms. The computed session key will be bound to the communication session. After the confirmation the authentic communication session is in place and both peers start to periodically send heartbeat messages.

State Change The heartbeat messages either confirm the current platform state or they report a state change. The reported platform state can reveal that the peer is under a DMA malware attack, that the suspicious peripheral could be stopped, or that no attack has been detected. If the peer stops sending heartbeat messages, the local platform assumes that the peer has been attacked by DMA malware executed on the network interface card. In this case, BARM has successfully terminated the ongoing DMA attack by stopping the network interface card. Depending on the local security policy a platform can tear down the channel, continue with the current session key, or renegotiate the channel. It is advantageous to continue with the current session key if the heartbeat message reports that the attack could be stopped immediately and if the local security policy states that this case is tolerable. To be more precise, it can make sense if the platform can continue to operate normally without the affected peripheral. In the case of an involved administrator platform, we expect that the administrator will analyze the attack in more detail as soon as possible to remove the DMA malware from the compromised peripheral or, if absolutely necessary, to exchange the compromised peripheral or chipset with a benign one.

3 Implementation of the Authentic Reporting Channel for BARM

BARM as presented in [1] is insufficient for the authentic channel based reporting application. When BARM sends network packets, it also causes bus activity that needs to be considered by BARM's detection model. To implement an authentic channel application for our DMA malware scenario we have (i) enhanced BARM's detection model, see Section 3.1 and (ii) modified the TLS protocol to bind BARM's measurement (state information) to that channel, see Section 3.2.

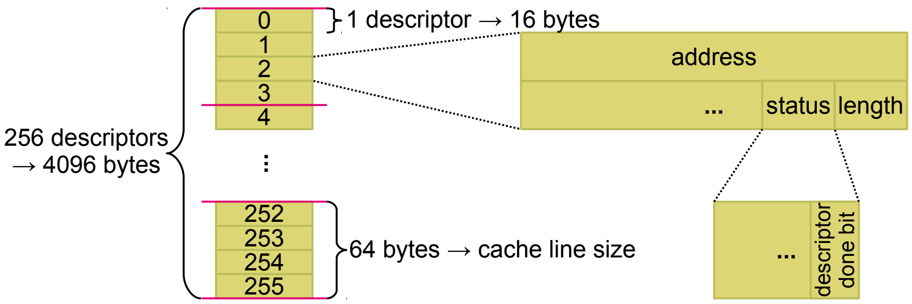
3.1 Bus Master Analysis: Ethernet Controller

To consider the ethernet controller in BARM's detection model we have to determine the expected bus activity value $\mathcal{A}_e^{ETH}$. Hence, we conducted a similar bus master analysis as presented in [1, Section 4.2] for the ethernet controller of our target platform. We analyzed the ethernet controller (namely *Ethernet Controller: Intel Corporation 82566DM-2 Gigabit Network Connection (rev 02)* [8]) of the same target platform as the previous experiments, see [7, 1]. The corresponding ethernet controller Linux device driver is `e1000e.ko`. To simplify our analysis we configured the driver to use legacy interrupts and no interrupt delays as well as no interrupt throttling. We also disabled checksumming and segmentation offloading for the network device.

The ethernet controller works with so-called descriptor rings, i. e., the transmit descriptor ring and the receive descriptor ring, see Figure 2. Each ring consists of 256 descriptors. A descriptor has a size of 16 bytes. This means that the device driver allocates 4096 bytes for each ring. If the host intends to send network packets, it prepares transmit descriptors and informs the ethernet controller that new descriptors are ready to be processed. The ethernet controller reads the descriptors via DMA from the host memory. After evaluating the descriptor the controller copies the network packet data from the host memory address that is present in the descriptor (see Figure 2) to its internal memory to be able to send the packet. If the ethernet controller has processed the descriptor, the controller "returns" the descriptor to the host by writing the `descriptor done bit` in the `status` field of the descriptor via DMA. When receiving network packets the process is similar except that the ethernet controller writes the network packet data into the host memory.

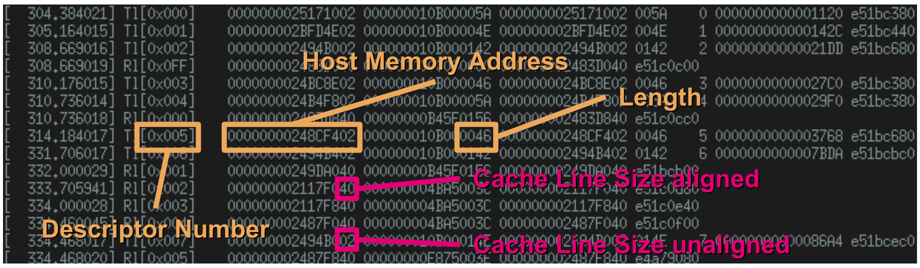
Cache Line Size To integrate the ethernet controller as a bus master into BARM's detection model we have to consider that the size of network packets is usually greater than keystroke codes, see [1, Section 4.2]. Keystroke codes are transfered via one bus transaction. This is not valid for network packets that have a size of 1514 bytes for example. To be able to determine how many bus transactions are necessary to transfer a particular amount of data we introduce a new parameter, i. e., the *cache line size*. The system cache is organized in cache

Figure 2: Transmit/Receive Descriptor Ring Structure



When the device driver informs the NIC that new network packets are ready to be transmitted, the ethernet controller reads transmit descriptors from the descriptor ring. The controller also reads the corresponding packets of the size that is stored in the `length` field of the descriptor from the host memory address that is stored in the `address` field of the descriptor. The ethernet controller writes the `descriptor done bit` in the `status` field of the descriptor if the the descriptor has been processed. When new network packets arrive from the network, the ethernet controller reads receive descriptors from the descriptor ring. Afterwards, the controller writes the corresponding packets of the size that is stored in the `length` field of the descriptor to the host memory address. The address is stored in the `address` field of the descriptor. The ethernet controller writes the `descriptor done bit` in the `status` field of the descriptor if the the descriptor has been processed.

Figure 3: Transmit Descriptor/Receive Descriptor Dump of the `e1000e.ko` Driver



```
[   304.384021] T1[0x000]   0000000025171002 000000010B00005A 0000000025171002 005A    0 0000000000001120 e51bc380
[   305.164015] T1[0x001]   000000002BFD4E02 000000010B00004E 000000002BFD4E02 004E    1 000000000000142C e51bc440
[   308.669016] T1[0x002]   000000002494B002 000000010B000142 000000002494B002 0142    2 00000000000021DD e51bc680
[   308.669019] R1[0x0FF]   0000000024[illegible] [illegible] [illegible]2483D040 e51c0c00
[   310.176015] T1[0x003]   0000000024BC8E02 0000000[illegible]B000046 0000000024BC8E02 0046    3 00000000000027C0 e51bc380
[   310.736014] T1[0x004]   0000000024B4F802 [illegible]0000010B00005A 0000000024[illegible]80[illegible]4 00000000000029F0 e51bc380
[   310.736018] R1[0x000]   0000000024[illegible]840 00000000B45F0156 00000[illegible]2483D840 e51c0cc0
[   314.184017] T1[0x005]   00000000248CF402 000000010B000046 00000000248CF402 0046    5 0000000000003768 e51bc680
[   331.706017] T1[0x006]   000000002494B402 000000010B000142 000000002494B402 0142    6 0000000000007BDA e51bcbc0
[   332.000029] R1[0x001]   00000000249DA040 00000000B45F015C 00000000249DA040 e51bcb00
[   333.705941] R1[0x002]   000000002117F040 000000004BA5003C 000000002117F040 e51c0d80
[   334.000028] R1[0x003]   000000002117F840 000000004BA5003C 000000002117F840 e51c0e40
[   334.460045] R1[0x004]   000000002487F040 000000004BA5003C 000000002487F040 e51c0f00
[   334.468017] T1[0x007]   000000002494B002 000000010B00014E 000000002494B002 014E    7 00000000000086A4 e51bcec0
[   334.468020] R1[0x005]   000000002487F840 00000000E875003E 000000002487F840 e4a79080
```

The dump reveals the most important information to derive the number of bus transactions caused by the ethernet controller. Some host memory addresses are not cache line size aligned. This can result in an additional bus transaction.

lines. Memory accesses are handled in cache lines of a certain cache line size $\mathcal{C} \in \mathbb{N}$ [see 9, p.223]. $\mathcal{C}$ is 64 bytes for our platform [see 10, p.17]. That means, if one word is requested from main memory, 64 bytes are actually transfered in one memory transaction. It is assumed that data that is adjacent in the host memory will likely be accessed in a subsequent operation. If so, these bytes are already in the cache and no additional transaction is needed. Memory access of peripherals is also handled in cache lines. It is possible that such a transaction must be snooped to ensure a coherent cache line [see 10, p.27].

The descriptor dump of the `e1000e.ko` driver depicts the host memory addresses of the network packet data, see Figure 3. The dump also reveals that not every address is cache line size aligned. This means that the number of bus transactions required to transfer the network packet data via DMA is not necessarily the value stored in the `length` field divided by the cache line size. Another important point relates to the receive descriptor handling. According to Intel [8] the ethernet controller optimizes the process of returning receive descriptors. That means, when receiving packets the ethernet controller does not write the `descriptor done bit` for each descriptor individually. Instead, it "collects" four descriptors that belong to the same cache line to be able to write four `descriptor done bits` with one bus transaction, see Figure 2. We consider both scenarios for the equation to compute the expected bus transactions caused by the ethernet controller.

Expected Bus Activity of the Ethernet Controller $\mathcal{A}_e^{ETH}$ (see [1, Section 3]) Due to our analysis we define the expected bus activity of the ethernet controller as follows:

$$\mathcal{A}_e^{ETH} = \mathcal{A}_e^{TX_{reads}} + \mathcal{A}_e^{TX_{writes}} + \mathcal{A}_e^{RX_{reads}} + \mathcal{A}_e^{RX_{writes}} \tag{1}$$

$\mathcal{A}_e^{TX_{reads}}$ is the expected bus activity that is caused by memory reads when transmitting a packet. $\mathcal{A}_e^{TX_{writes}}$ represents activity that is caused by memory writes. Analogously, $\mathcal{A}_e^{RX_{reads}}$ and $\mathcal{A}_e^{RX_{writes}}$ are introduced to consider the bus activity when receiving network packets. To compute $\mathcal{A}_e^{TX_{reads}}$, $\mathcal{A}_e^{TX_{writes}}$, $\mathcal{A}_e^{RX_{reads}}$, and $\mathcal{A}_e^{RX_{writes}}$ for one BARM sampling interval we have to consider the cache line size for the memory buffers that are read and written. That means for the memory buffer that stores the network packet data in host memory we have to align the memory buffer start address, which is stored in the `address` field ($hma \in \mathbb{N}$) of a descriptor, to the previous cache line size aligned address. The result is $ba\_start \in \mathbb{N}$:

$$ba\_start \;=\; hma \;-\; (hma \bmod \mathcal{C}) \tag{2}$$

The alignment for the memory buffer end address ($ba\_end \in \mathbb{N}$), which is the sum of the value in the `address` field (hma) and the value of the `length` field ($len \in \mathbb{N}$) of a descriptor is as follows:

$$ba\_end \;=\; hma \;+\; len \;+\; \mathcal{C} \;-\; ((hma \;+\; len) \bmod \mathcal{C}) \tag{3}$$

The same alignment is required for descriptor transfers. The transfer start address is determined by the descriptor number of the last descriptor of the previous sampling interval ($old\_d \in \mathbb{N}$). The transfer end address is determined by the descriptor number of the last descriptor of the current sampling interval

($cur\_d \in \mathbb{N}$). When considering the cache line size the alignment results in descriptor numbers $d\_start \in \mathbb{N}$ and $d\_end \in \mathbb{N}$ as follows ($\mathcal{D} \in \mathbb{N}$ is the descriptor size in bytes, i. e., 16 bytes in our case):

$$d\_start \;=\; old\_d \;-\; \frac{((old\_d \;\times\; \mathcal{D}) \bmod \mathcal{C})}{\mathcal{D}} \tag{4}$$

$$d\_end \;=\; \frac{cur\_d \;\times\; \mathcal{D} \;+\; \mathcal{C} \;-\; ((cur\_d \;\times\; \mathcal{D}) \bmod \mathcal{C})}{\mathcal{D}} \tag{5}$$

For one sampling interval $\mathcal{A}_e^{TX_{reads}}$, $\mathcal{A}_e^{TX_{writes}}$, $\mathcal{A}_e^{RX_{reads}}$, and $\mathcal{A}_e^{RX_{writes}}$ are computed as follows:

$$\mathcal{A}_e^{TX_{reads}} \;=\; \sum_{n=1}^{cur\_d^{TX}-old\_d^{TX}} \left(1 \;+\; \frac{ba\_end_n^{TX} \;-\; ba\_start_n^{TX}}{\mathcal{C}}\right) \tag{6}$$

It is necessary to add 1 memory read bus transaction for each transmit descriptor because of the corresponding descriptor fetch that is (according to our experiments) not optimized in terms of cache lines. This is handled differently when writing the `descriptor done bit`. In this case the ethernet controller tries to write as many `descriptor done bits` as possible. The maximum is four bits for one bus transaction.

$$\mathcal{A}_e^{TX_{writes}} \;=\; \frac{(d\_end^{TX} \;-\; d\_start^{TX}) \;\times\; \mathcal{D}}{\mathcal{C}} \tag{7}$$

When receiving network packets, memory reads only occur due to receive descriptor fetching. We determined that the ethernet controller fetches four receive descriptors (equals to the cache line size) with one memory read bus transaction during our experiments. We use the indicator function with $N := \{n \;\in\; [old\_d^{RX},\, cur\_d^{RX}] \mid (n \;\times\; \mathcal{D} \bmod \mathcal{C}) \;=\; 0\}$ in the following equation:

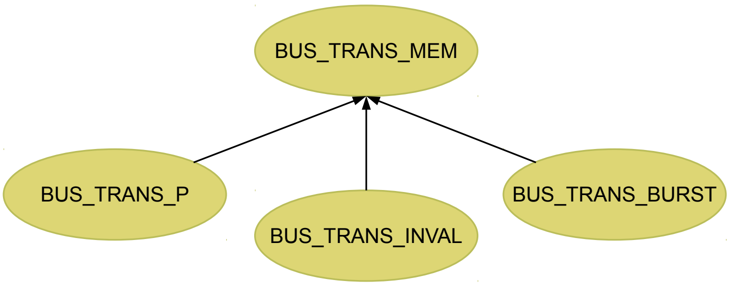
$$\mathcal{A}_e^{RX_{reads}} \;=\; \sum_{n=old\_d^{RX}}^{cur\_d^{RX}} \mathbb{1}_N(n) \tag{8}$$

The number of expected bus transactions due to memory writes are as follows:

$$\begin{aligned} \mathcal{A}_e^{RX_{writes}} \;=\; & \sum_{n=1}^{cur\_d^{RX}-old\_d^{RX}} \frac{ba\_end_n^{RX} \;-\; ba\_start_n^{RX}}{\mathcal{C}} \\ & + \frac{(d\_end^{RX} \;-\; d\_start^{RX}) \;\times\; \mathcal{D}}{\mathcal{C}} \end{aligned} \tag{9}$$

We expect that network packet data must be copied to the host memory and that corresponding `descriptor done bits` will be written to the descriptors in the host memory.

Figure 4: BUS_TRANS Event Counter



The sum of BUS_TRANS_BURST, BUS_TRANS_P and BUS_TRANS_INVAL counts results in BUS_TRANS_MEM counts [11].

Exploiting Additional BUS_TRANS Events We verified Equation 1 with further BUS_TRANS event counter (see Figure 4) that are basically subsets of the event BUS_TRANS_MEM. We determined that the event counter BUS_TRANS_P counts the memory reads of a peripheral and that the event counter BUS_TRANS_INVAL counts the memory writes of a peripheral. We used these counters in conjunction with THIS_AGENT and ALL_AGENTS name extensions as described in [1, Section 4.2] to distinguish bus transactions caused by the host CPU and bus transactions caused by the peripheral. The event BUS_TRANS_BURST did not occur during our experiments. The number of bus transactions caused by the ethernet controller is computed according to Equation 1 when the e1000e.ko driver function e1000_clean_tx_irq or e1000_clean_rx_irq is called. We enhanced BARM as introduced in [1] to consider $\mathcal{A}_e^{ETH}$ as described in this section.

3.2 Implementation based on OpenSSL

OpenSSL is a popular software toolkit that implements cryptographic mechanisms such as the SSL/TLS protocol and the encoding/decoding of X.509 certificates. The toolkit provides the developer with shared libraries, i. e., libssl and libcrypto. The openssl command line tool also makes use of these libraries. Applications that require the cryptographic mechanisms provided by OpenSSL can use the libraries directly. Note, the implementation presented in this section is based on our [3] previous trusted channel implementation. Our modifications are based on TLS and TLS related *Request for Comments* (RFC) documents, i. e., RFC4366 and RFC4680. Hence, the modifications are compliant with the TLS specification.

The TLS handshake protocol used to negotiate a session key of a secure channel needs to be adapted to consider BARM's measurement results. Considering the measurement results during the handshake enables the peer to determine if the target platform is already attacked by DMA malware. This helps the peer to decide if the target platform is trustworthy. The peer can abort the handshake of the authentic reporting channel if the other endpoint is considered untrustworthy. Note, due to our trust model we consider the host CPU as a channel endpoint. Other computing environments including the network interface card do not belong to the endpoint. We use asymmetric cryptography mechanisms and certificates to authenticate endpoints. In the following paragraphs we describe the used key exchange and certificate. We also describe extensions for the TLS *Hello* messages. Extensions to the TLS protocol are considered by Dierks and Rescorla [12]. To transmit BARM measurement results (platform state data) additional handshake messages are required. We use *Supplemental Data* messages for this purpose.

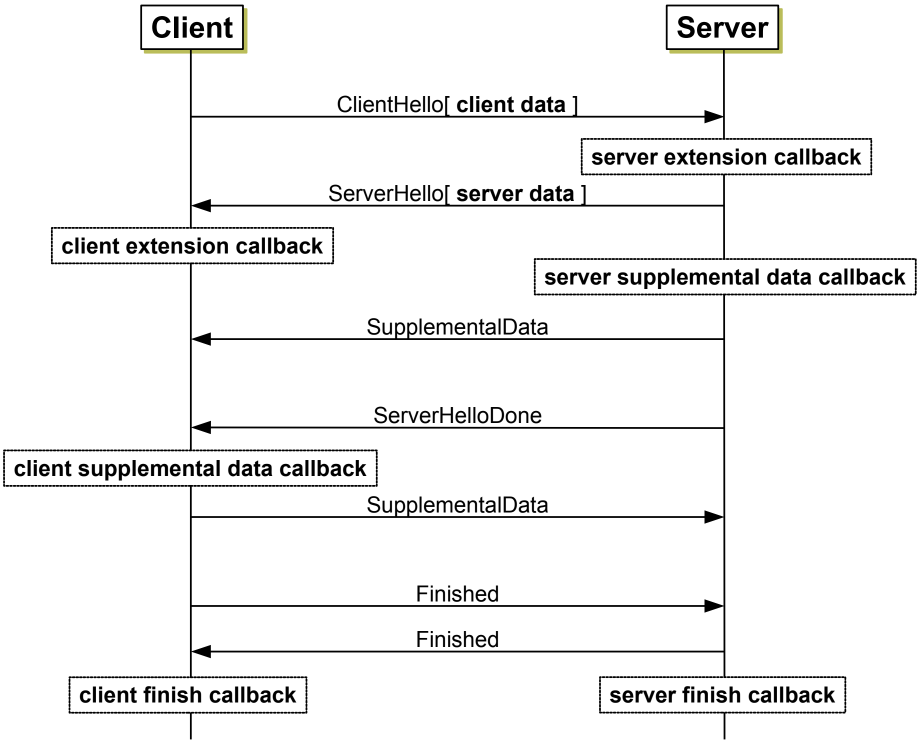
Key Exchange Type Our implementation of the authentic report channel is based on an adapted version of the TLS Diffie-Hellman Ephemeral RSA (DHE-RSA) handshake.[2] That means, to authenticate endpoint data an RSA signing key pair is used. For the negotiation of the session key Diffie-Hellman values are used. The public Diffie-Hellman part that is transmitted to the peer is signed by the secret part of the RSA signing key pair.

Endpoint Certificate To authenticate the endpoints, certificates (see *cert* in Figures 6 and 7) are exchanged during the TLS handshake. When using DHE-RSA, the certificates exchanged via *Certificate* messages contain the public part PK_{sign} of the signing key pair $K_{sign} := (SK_{sign}, PK_{sign})$. We have to ensure that the secret key SK_{sign} is only available to the endpoint. Our certificates include a BARM related identifier to *bind* the TLS-based authentic reporting channel to the endpoint. A certificate that includes a BARM identifier is issued by a trusted third party that vouches for a correct BARM installation on the target platform and that the secret key part SK_{sign} is only available on that endpoint. Hence, the certificate *cert* links the signing key K_{sign} to the endpoint that executes BARM. K_{sign} key pairs must be used to authenticate data sent by the client C and server S during the handshake. This eventually binds the transmitted platform state data to the authentic reporting channel. The trusted third party that vouches for the correct BARM installation and for the secret signing key part SK_{sign} could be the administrator who also runs the evaluation platform that receives platform state data (BARM measurements) from the target platform. The used certificate is actually a normal TLS certificate that includes the BARM related identifier. The certificate as well as the signing key pair K_{sign} are deployed together with BARM and are considered as long-lived.

Modifications to Hello Messages We use the *ClientHello* and *ServerHello* messages to negotiate the security parameters of the authentic reporting channel, see Figure 6. The client platform C that runs BARM starts the adapted TLS client and sends the *ClientHello* message to the server platform S. The server replies with *ServerHello*. The *Hello* messages include the security parameters *sec_param* (see Section 2) of the corresponding peer, see Figure 6. The security parameters determine which endpoint has to provide platform state data, i. e., BARM measurements. We use *Hello message extensions* [12] to exchange security parameters. Our OpenSSL-based implementation makes use of the *TLS Hello Extensions* as described in RFC4366 [14]. A patch for OpenSSL (0.9.8.x)

[2] As described in prior work [3] other key exchange methods such as RSA and DH-RSA can also be used to implement a trusted channel based authentic reporting application.

Figure 5: TLS Handshake Considering Hello Extensions and Supplemental Data Extensions


Client
Server
ClientHello[client data]
server extension callback
ServerHello[server data]
client extension callback
server supplemental data callback
SupplementalData
ServerHelloDone
client supplemental data callback
SupplementalData
Finished
Finished
client finish callback
server finish callback

The *ClientHello* message contains `client data` **and the *ServerHello* message contains** `server data`. **Additional *SupplementalData* messages contain** `client supplemental data` **and** `server supplemental data`. **Supplemental data is also considered as TLS extension. (based on [13])**

implements the hello extensions, see Figure 5.[3] The patch modifies code related to the library `libssl`. We use this patch for our authentic reporting channel application implementation.

The patch provides an interface that allows the developer to register new TLS extensions [see 13]. A TLS extension that is represented by the `TLSEXT_GENERAL` object transmits generic data. The application that uses TLS specifies the data format of the generic data. TLS extensions consist of a type, the data length, and the generic data (type-length-value format) as well as certain flags[4] and callback functions that implement the required extension logic. Callbacks (see Figure 5) are only triggered on the peer that instantiated the corresponding `TLSEXT_GENERAL` object. The generic data that is transmitted via a *Hello* message is one generic datum. In our implementation the TLS extension that is exchanged via *Hello* messages (hello extension) is:

- `ARCH_NEGOTIATION_EXT`: This extension (`EXT`) for our authentic reporting channel (`ARCH`) is used to negotiate security parameters sec_param.

Client as well as server register hello extensions (`TLSEXT_GENERAL` objects) if they want to handle them. If a peer receives a *Hello* message that contains the registered extension, the peer calls the corresponding extension callback, see Figure 5.

Supplemental Data Messages for Platform State Data The client platform C as well as the server platform S can provide platform state data. We use so-called *SupplementalData* messages (see Figure 5) as specified by the *Internet Engineering Task Force Networking Group* in RFC4680 [15] to transmit platform state data. The OpenSSL patch also implements *SupplementalData* messages for OpenSSL (0.9.8.x).[5] The details of the implementation of this patch are explained by Davide Vernizzi [13]. As described in RFC4680 supplemental data is also used to transmit generic data. The peer determines whether or not the generic data needs to be transmitted using hello extensions. The OpenSSL patch also enables us to define supplemental data extensions that we need for our authentic reporting channel. Supplemental data extensions also consist of a type, the generic data, the data length, and callback functions. Supplemental data transmitted using the *SupplementalData* message can be a stack of several generic data. In our implementation the extensions to exchange generic data via *SupplementalData* messages are:

- `ARCH_SUPP_DATA_C_EXT`: This extension is used to transmit the platform state data PSD^C (supplemental data) from the client C to the server S.

- `ARCH_SUPP_DATA_S_EXT`: This extension is used to transmit the platform state data PSD^S (supplemental data) from the server S to the client C.

[3] The TLS hello extensions and supplemental data patch can be found at `http://openssl.6102.n7.nabble.com/PATCH-TLS-hello-extensions-and-supplemental-data-td38202.html` [accessed 25 February 2014]

[4] The extension flags are `client_required` (the client will abort if the server ignores the extension where this flag has been set), `server_send` (the server will send the extension where this flag has been set), and `received` (internal use, e. g., to check duplicates).

[5] See Footnote 3

The patched OpenSSL software handles generic data as presented in Figure 5. Callback functions that also belong to the TLS extensions are called to process the generic data according to the required extension logic. Analogous to hello extensions, client and server have to register for supplemental data extensions that they want to handle via the corresponding supplemental data callbacks. Figure 6 and 7 depict how our generic data (hello extensions as well as supplemental data extensions) is handled using the callback functions during the adapted TLS handshake.

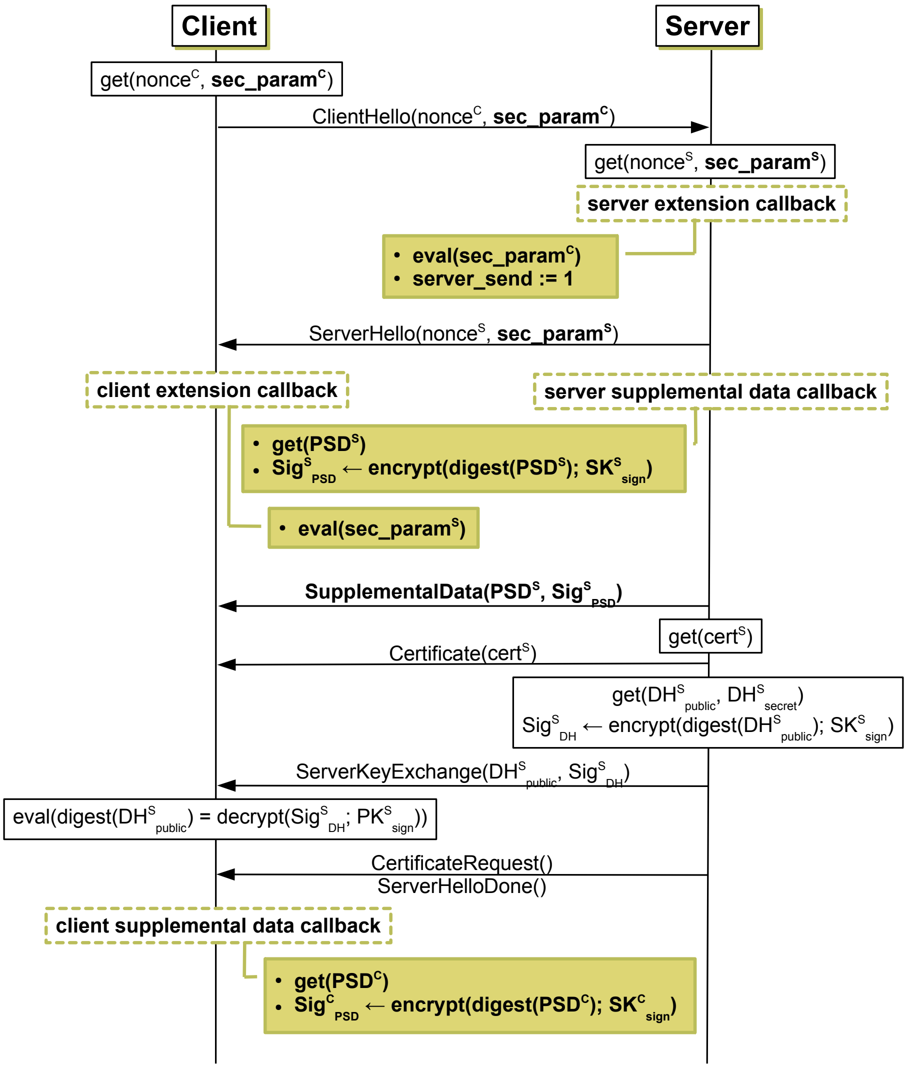
In our proof of concept implementation the generic data format used to exchange platform state data PSD via supplemental data is quite simple:

- `barm_measurement`: This data field contains the BARM measurement taken by the BARM Linux kernel module.
- Devices flag pair list: We use a devices flag pair list to communicate if a peripheral is attacking the target platform. The first flag represents if the corresponding device started to attack the host and, if so, the second flag states if the malicious device could be stopped. The devices flag pair list looks as follows:
  - `(uhci_attack, uhci_disabled)`: This flag pair represents the *Universal Host Controller Interface Controller* (UHCI, see [1, Section 4.2]) controller.
  - `(..._attack, ..._disabled)`: *[further devices]*
  - `(me_attack, me_disabled)`: This flag pair represents the manageability engine.
- *nonceSD*: *nonceSD* consists of the two elements:
  - $nonce^C$ (`client_random`)
  - $nonce^S$ (`server_random`)

The signature Sig_{PSD} on the platform state data PSD is also sent to the peer via the *SupplementalData* message, see Figure 6 and 7. By doing so, the platform state data PSD is also bound to the corresponding secure channel. The *nonceSD* included in the supplemental data is compared with $nonce^C$ and $nonce^S$ (sent via the *Hello* messages) to guarantee freshness of the received platform state data PSD. To authenticate and to be able to check the integrity of platform state data PSD, we use the secret part SK_{sign} of the signing key pair to sign PSD. To be able to verify the signature each peer provides the certificate that contains the public key part PK_{sign} using the *Certificate* message directly after transmitting the *SupplementalData* message, see Figures 6 and 7. The BARM measurement results that are also part of the supplemental data are evaluated to derive the trustworthiness of the peer. Depending on the derived trustworthiness the local platform takes measures according to the local security policy.

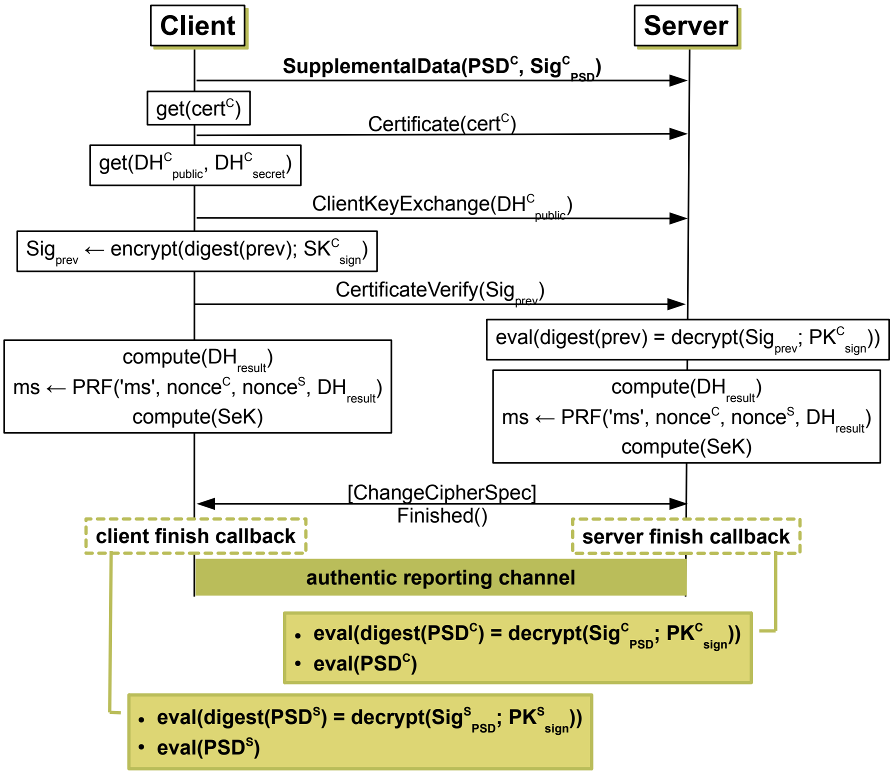
Session Key Computation The session key SeK is computed on both peers as usual. Since we use DHE-RSA, the secure channel that uses SeK is eventually linked to the endpoints (host CPUs). The exchanged DH parts are signed using

Figure 6: Adapted TLS-DHE-RSA Handshake for the Authentic Reporting Channel (a)


Client
Server
get(nonce^C, **sec_param^C**)
ClientHello(nonce^C, **sec_param^C**)
get(nonce^S, **sec_param^S**)
server extension callback
• **eval(sec_param^C)**
• **server_send := 1**
ServerHello(nonce^S, **sec_param^S**)
client extension callback
server supplemental data callback
• **get(PSD^S)**
• **Sig$^S_{PSD}$ ← encrypt(digest(PSD^S); SK$^S_{sign}$)**
• **eval(sec_param^S)**
SupplementalData(PSD^S, Sig$^S_{PSD}$)
get(cert^S)
Certificate(cert^S)
get(DH$^S_{public}$, DH$^S_{secret}$)
Sig$^S_{DH}$ ← encrypt(digest(DH$^S_{public}$); SK$^S_{sign}$)
ServerKeyExchange(DH$^S_{public}$, Sig$^S_{DH}$)
eval(digest(DH$^S_{public}$) = decrypt(Sig$^S_{DH}$; PK$^S_{sign}$))
CertificateRequest()
ServerHelloDone()
client supplemental data callback
• **get(PSD^C)**
• **Sig$^C_{PSD}$ ← encrypt(digest(PSD^C); SK$^C_{sign}$)**

Modifications that were made to the TLS handshake are highlighted in bold text. The adapted handshake is continued in Figure 7.

Figure 7: Adapted TLS-DHE-RSA Handshake for the Authentic Reporting Channel (b)


Client
Server
SupplementalData(PSD^C, Sig^C_PSD)
get(cert^C)
Certificate(cert^C)
get(DH^C_public, DH^C_secret)
ClientKeyExchange(DH^C_public)
Sig_prev ← encrypt(digest(prev); SK^C_sign)
CertificateVerify(Sig_prev)
eval(digest(prev) = decrypt(Sig_prev; PK^C_sign))
compute(DH_result)
ms ← PRF('ms', nonce^C, nonce^S, DH_result)
compute(SeK)
compute(DH_result)
ms ← PRF('ms', nonce^C, nonce^S, DH_result)
compute(SeK)
[ChangeCipherSpec]
Finished()
client finish callback
server finish callback
authentic reporting channel
eval(digest(PSD^C) = decrypt(Sig^C_PSD; PK^C_sign))
eval(PSD^C)
eval(digest(PSD^S) = decrypt(Sig^S_PSD; PK^S_sign))
eval(PSD^S)

After the handshake has been finished the authentic reporting channel is used by BARM to transmit heartbeat messages in a regular interval to communicate platform state changes, i. e., to report a DMA malware based attack.

the secret part of K_{sign} (SK_{sign}) that links the DH values to K_{sign}. The signing key pair K_{sign} is bound to exactly one endpoint due to the certificate issued by the trusted third party that vouches for the fact that SK_{sign} is only available on the endpoint. Hence, the session key is also bound to the endpoint.

Heartbeat Messages After the handshake has been completed, BARM uses the negotiated channel to send heartbeat messages in a regular interval to the external administrator platform. These messages contain the current BARM measurement and the devices flag pair list in a similar PSD format that has been used during the handshake. Only *nonceSD* is missing. The regular heartbeat messages are used by BARM to report platform state changes, i. e., a DMA malware based attack. If the external platform does not receive heartbeat messages anymore we assume that the NIC tried to attack the host platform and BARM was able to successfully stop the attack. It is also possible that malware that is executed on the NIC blocks the heartbeat messages. If so, the attack is also revealed.

4 Evaluation

We use the same platform and basic evaluation configuration as described in [1, Section 5] to evaluate the enhanced BARM. Please note, only the client platform must transmit platform state data.

4.1 Expected Bus Activity Validation

To validate Equation 1 we conducted different tests. The evaluation results are depicted in Figure 8. The results reveal larger fluctuations in BARM measurement results when the `ping`, `scp` and `wget` command cause network traffic. Table 1 provides information on the cause of the larger fluctuations. The table presents BARM measurements that were taken during the download of a 1 GB file using the `wget` command. The applied sampling interval was 32 ms. The table depicts that a larger positive discrepancy (see BARM sample 125924: 13 bus transactions) is followed by a larger negative discrepancy (see BARM sample 125925: -12 bus transactions). We assume that a positive discrepancy occurs when network packets were already copied to the host memory by the ethernet controller, but BARM was unable to evaluate the corresponding receive descriptors in the current sampling interval. These descriptors are available in the next sampling interval. Hence, BARM evaluates the descriptors in the next interval, which in turn results in a negative discrepancy. BARM subtracts expected bus transactions from the measured transactions that were actually measured in the last sampling interval.

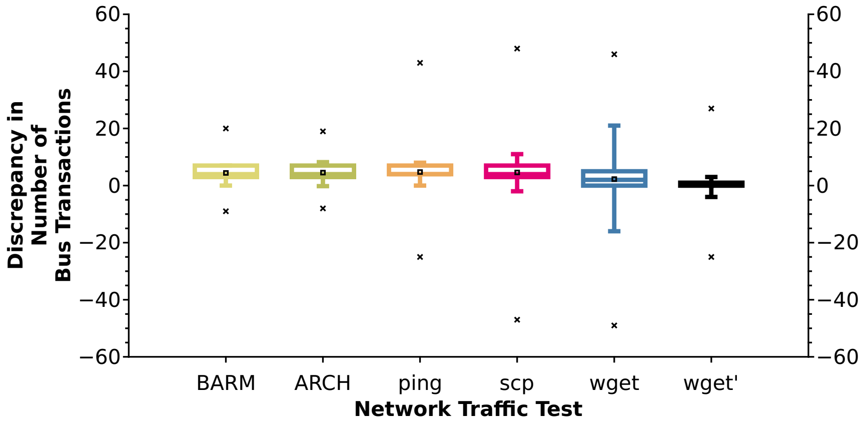
As depicted in Table 1, the positive and negative values compensate one another. Thus, the fluctuation can be minimized by simply adding positive and negative BARM measurement values. As presented in Table 1, a pair of positive and negative measurement values can also occur the other way around (see BARM samples 125926 and 125927, for example). This means that the negative value is determined before the positive value. We assume that this occurs when BARM has already analyzed transmit descriptors when the corresponding packets were not copied by the ethernet controller yet. Hence, BARM already

Table 1: BARM Measurement Values Revealing Fluctuations

BARM sampling number	BARM measurement value	BARM sampling number	BARM measurement value
125912	5	125944	2
125913	2	125945	25
125914	3	125946	-48
125915	2	125947	28
125916	3	125948	0
125917	0	125949	3
125918	0	125950	5
125919	1	125951	1
125920	2	125952	13
125921	-17	125953	-21
125922	22	125954	5
125923	1	125955	2
125924	13	125956	2
125925	-12	125957	-1
125926	-15	125958	4
125927	22	125959	3
125928	5	125960	-21
125929	0	125961	25
125930	-2	125962	2
125931	5	125963	-2
125932	2	125964	3
125933	5	125965	2
125934	0	125966	5
125935	-2	125967	2
125936	9	125968	-1
125937	2	125969	2
125938	3	125970	2
125939	-2	125971	6
125940	8	125972	0
125941	-3	125973	1
125942	0	125974	2
125943	5	125975	3

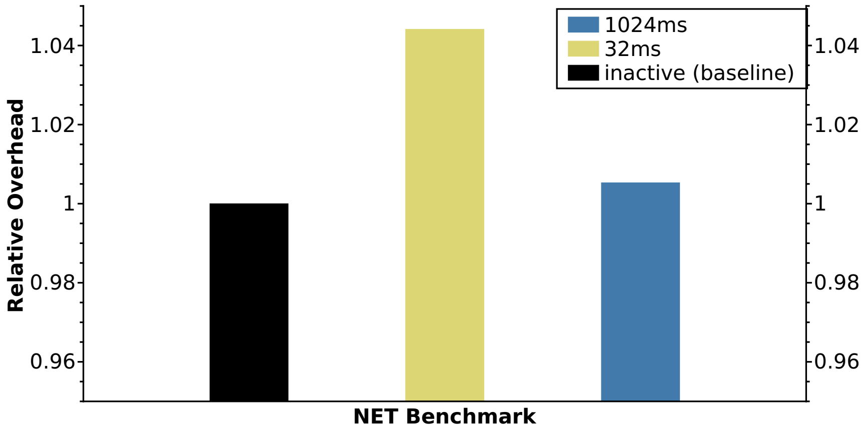
The sampling numbers and the corresponding measurement values taken are from the measurement log that was taken when downloading a 1 GB **file from** `http://download.thinkbroadband.com/1GB.zip` **[accessed 25 February 2014]. The BARM sampling interval was** 32 ms**.**

Figure 8: Expected Bus Activity Evaluation with Network Traffic



We evaluated the expected bus activity for six different test cases. The discrepancy is visualized in the form of boxplots. In the first case (**BARM**) we only run the enhanced BARM and in the second case we run the enhanced BARM together with the OpenSSL-based authentic reporting channel. We took 100 BARM measurements in both cases. BARM and the authentic reporting channel are also active in the remaining test cases (**ping**, **scp**, **wget**, **wget'**). We executed the `ping` command with a 1000 bytes payload 100 times (**ping**). In the case of **scp** we copied a 100 MB file from an external platform to our target platform 100 times. In the **wget** case we downloaded a 1 GB file from `http://download.thinkbroadband.com/1GB.zip` [accessed 25 February 2014] using the `wget` command. We applied a BARM sampling interval of 32 ms for all test cases except the last one (**wget'**). The boxplot for the **wget'** case represents the result when using a sampling interval of 1024 ms during a `wget` download of a 1 GB file.

Figure 9: Relative Performance Overhead for Different Reporting Intervals and Constant Sampling Interval



The figure compares the results of three measurement series. The first measurement series (inactive) represents the baseline. Inactive means that BARM was not running and no heartbeat messages were sent. A bar in the figure represents the mean of 100 measurements. We measured the clock cycles (with time stamp counters) that are needed to copy a 100 MB file from an external platform with the `scp` command. Measurements were taken for a reporting interval of 32 ms and for a reporting interval of 1024 ms. In both cases we used the same BARM sampling interval of 32 ms. The relative performance overhead when sending a heartbeat message every 32 ms is approximately 4.5 %. The overhead is only approximately 0.5 % when sending the message every 1024 ms.

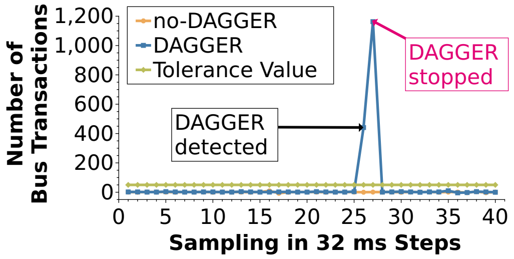
subtracts the expected bus transactions from the measured ones before they are actually measured. The transactions are measured in the next sampling interval that results in a larger positive discrepancy.

We examined the described behavior with two sampling intervals when using the `wget` command to download a 1 GB file. As depicted in Figure 8, the fluctuations are larger when using `wget` with a sampling interval of 32 ms (see wget) compared to a sampling interval of 1024 ms (see wget').

4.2 Network Performance Overhead Evaluation

We conducted a network benchmark to reveal the network performance overhead that is caused by the enhanced BARM version. The enhanced BARM version permanently sends heartbeat messages. The results are presented in Figure 9. The results in Figure 9 reveal a relative performance overhead of approximately 4.5 % when sending the heartbeat message every 32 ms. This interval length corresponds the the BARM sampling interval. It is not necessary to use the

Figure 10: Evaluating Enhanced BARM at an Arbitrary Point during Runtime with the Authentic Reporting Channel



The conducted experiment is similar to the experiment presented in [1, Section 5.3]. BARM's sampling interval was 32 ms **and the tolerance value was** 50 **bus transactions. This time BARM considers the ethernet controller as an additional bus master that allowed us to start our authentic reporting channel. Heartbeat messages were sent every** 32 ms**. The figure compares three curves, i. e., the tolerance value $\mathcal{T}$, BARM's measurement results without any attack, and BARM's measurement results with a DAGGER attack.**

same interval for reporting as for BARM measurement sampling due to the heartbeat message format that we use to transmit platform state data. The devices flag pair list represents a history of malicious peripherals. Hence, the network performance overhead for 32 ms sampling and reporting interval can be avoided. The only requirement is that the sampling interval is less or equal than the reporting interval.

4.3 Test with DAGGER

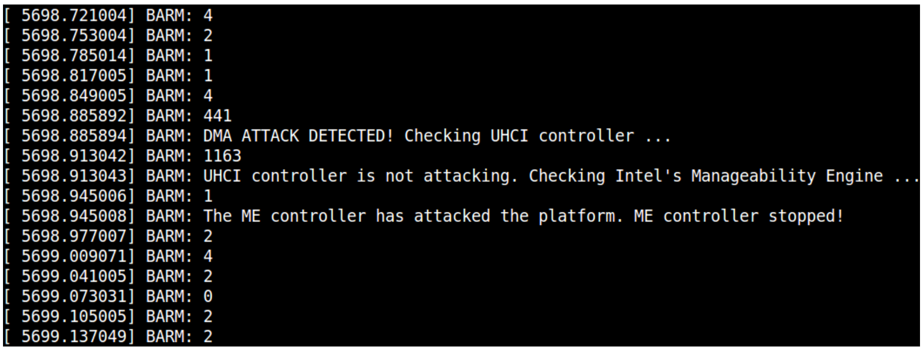
We repeated the DMA malware DAGGER[6] test (see [1]) with our enhanced bus agent runtime monitor BARM. The results are summarized in Figure 10, Figure 11, and Figure 12. We attacked the target platform at an arbitrary point in time during runtime. Figure 10 confirms that the enhanced BARM could reveal the DMA attack as well as stop the malicious peripheral. The excerpt from the log in Figure 11 and Figure 12 belong to the same experiment that was the basis for Figure 10.

5 Security Considerations

In this section we informally evaluate the security requirements that we introduced in the beginning of this technical report. A formal proof is outside the scope of this work. Many research related to security proofs of the TLS protocols have been published in the past. An overview is presented by Kohlweiss et al. [16]. The research also considers multiple TLS variants. We assume that

[6] The *Direct Memory Access based keystroke code loGGER* DAGGER is presented in [7].

Figure 11: BARM Authentic Reporting Channel – Client Side



```
[ 5698.721004] BARM: 4
[ 5698.753004] BARM: 2
[ 5698.785014] BARM: 1
[ 5698.817005] BARM: 1
[ 5698.849005] BARM: 4
[ 5698.885892] BARM: 441
[ 5698.885894] BARM: DMA ATTACK DETECTED! Checking UHCI controller ...
[ 5698.913042] BARM: 1163
[ 5698.913043] BARM: UHCI controller is not attacking. Checking Intel's Manageability Engine ...
[ 5698.945006] BARM: 1
[ 5698.945008] BARM: The ME controller has attacked the platform. ME controller stopped!
[ 5698.977007] BARM: 2
[ 5699.009071] BARM: 4
[ 5699.041005] BARM: 2
[ 5699.073031] BARM: 0
[ 5699.105005] BARM: 2
[ 5699.137049] BARM: 2
```

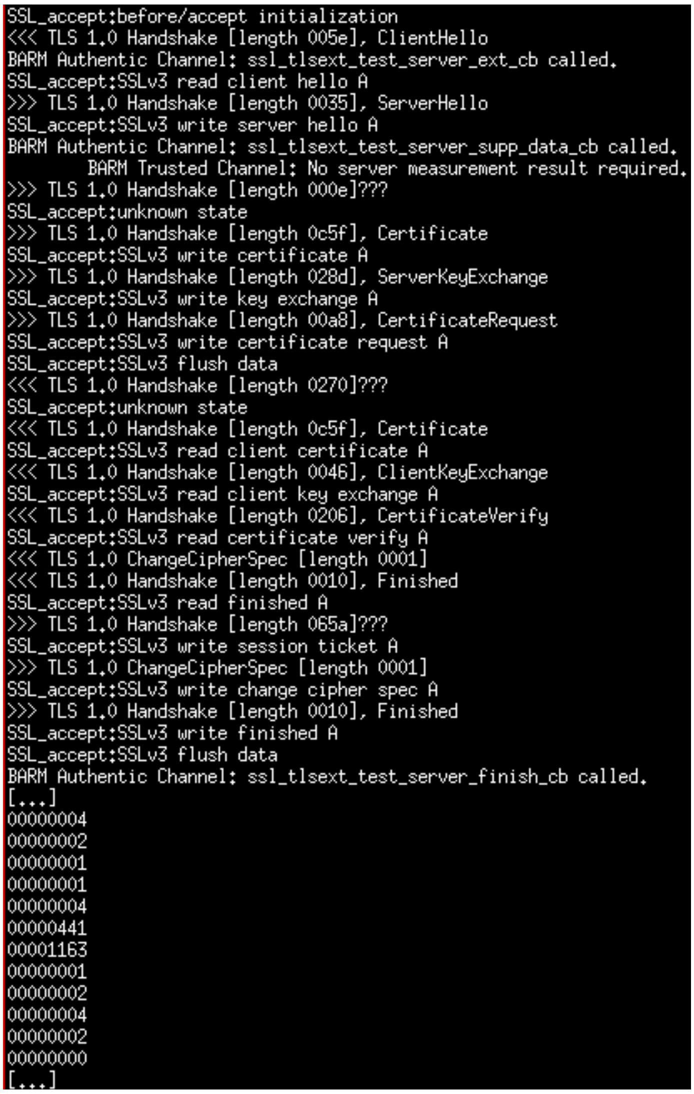
The figure presents a part of BARM's log output. BARM is deployed on the target platform, i. e., the client. The log output demonstrates that BARM revealed a DMA attack and that BARM was able to stop the malicious peripheral.

our TLS-based channel can also be formally proven. However, the focus of this work is the enhanced BARM that considers the network interface card. Hence, we review the extent to which our enhanced BARM fulfills the requirements for a secure channel (R1), binding of BARM measurements to the secure channel (R2), and privacy (R3).

- **R1 – Secure channel properties:** Due to the applied TLS protocol the secure channel properties confidentiality, integrity, authenticity as well as freshness are ensured for the communication channel. Due to the enhanced BARM these properties are also ensured on the endpoint, i. e., the host CPU. Given that the attacker has to search for valuable data, BARM ensures the integrity and the confidentiality of data that is present in the main memory. The attacker could merely randomly write to or read from the main memory without searching for valuable data. The attacker also needs to search for nonces, key material or the session key SeK as well as the private part of the signing key pair SK_{sign} to attack the communication session. Hence, the enhanced BARM also takes care of the properties authenticity and freshness on the endpoint due to the detection of additional bus traffic when the attacker searches in the main memory.

  The attacker can only conduct a MitM attack if the attacker is able to steal private key material or the session key via DMA. Scanning the memory for this data will be detected by BARM. BARM can also identify the malicious device. Hence, the access to the main memory can be prevented. Note, the host CPU could enforce the attacker to cause more bus transactions by storing parts of the sensitive data in processor registers. This technique was proposed in [17], for example. This will not protect the sensitive data, since DMA attacks can be used to dump the content of processor registers into the main memory. However, such an attack will cause more bus activity, which will also be detected by BARM.

Figure 12: BARM Authentic Reporting Channel – Server Side



```
SSL_accept:before/accept initialization
<<< TLS 1.0 Handshake [length 005e], ClientHello
BARM Authentic Channel: ssl_tlsext_test_server_ext_cb called.
SSL_accept:SSLv3 read client hello A
>>> TLS 1.0 Handshake [length 0035], ServerHello
SSL_accept:SSLv3 write server hello A
BARM Authentic Channel: ssl_tlsext_test_server_supp_data_cb called.
        BARM Trusted Channel: No server measurement result required.
>>> TLS 1.0 Handshake [length 000e]???
SSL_accept:unknown state
>>> TLS 1.0 Handshake [length 0c5f], Certificate
SSL_accept:SSLv3 write certificate A
>>> TLS 1.0 Handshake [length 028d], ServerKeyExchange
SSL_accept:SSLv3 write key exchange A
>>> TLS 1.0 Handshake [length 00a8], CertificateRequest
SSL_accept:SSLv3 write certificate request A
SSL_accept:SSLv3 flush data
<<< TLS 1.0 Handshake [length 0270]???
SSL_accept:unknown state
<<< TLS 1.0 Handshake [length 0c5f], Certificate
SSL_accept:SSLv3 read client certificate A
<<< TLS 1.0 Handshake [length 0046], ClientKeyExchange
SSL_accept:SSLv3 read client key exchange A
<<< TLS 1.0 Handshake [length 0206], CertificateVerify
SSL_accept:SSLv3 read certificate verify A
<<< TLS 1.0 ChangeCipherSpec [length 0001]
<<< TLS 1.0 Handshake [length 0010], Finished
SSL_accept:SSLv3 read finished A
>>> TLS 1.0 Handshake [length 065a]???
SSL_accept:SSLv3 write session ticket A
>>> TLS 1.0 ChangeCipherSpec [length 0001]
SSL_accept:SSLv3 write change cipher spec A
>>> TLS 1.0 Handshake [length 0010], Finished
SSL_accept:SSLv3 write finished A
SSL_accept:SSLv3 flush data
BARM Authentic Channel: ssl_tlsext_test_server_finish_cb called.
[...]
00000004
00000002
00000001
00000001
00000004
00000441
00001163
00000001
00000002
00000004
00000002
00000000
[...]
```

The figure depicts the log output of the adapted OpenSSL server. The log consists of the TLS handshake messages, callback call messages, and received BARM measurements. The measurement values are the same as presented in Figure 11. The BARM instance that is deployed on the client side was able to stop the attack. In this example, the local security policy tolerates the stopped attack. Alternatively, the server could have torn down the channel when the server received the BARM measurement of 441 bus transactions. The server was also configured with $\mathcal{T} = 50$ bus transactions.

The attacker could attempt to modify BARM measurements. To do so, the attacker could try to find the variables in the main memory where BARM stores the values of the performance monitoring units that we exploit to reveal DMA attack. However, the DMA-based search would be revealed by BARM. Alternatively, the attacker could try to modify the host CPU registers that correspond to the performance monitoring units used by BARM. The attacker has no direct access to host CPU registers. However, the attacker needs to find a memory area to store host CPU instructions that modify the performance monitoring processor registers. It is required that the host CPU will sooner or later consider the memory area, which contains the malicious instructions. Again, the attacker has to search for such an area via DMA and this DMA-based search will be revealed by BARM.

- **R2 – Binding of BARM measurements to the secure channel:** Authenticity of an endpoint is ensured by providing the certificate $cert$ that includes the BARM identifier and the public key part of the signing key pair PK_{sign}. The certificate is signed by a trusted party. Two factors ensure that the BARM measurements are bound to the channel. First, the BARM measurement that is transmitted during the handshake is signed using the endpoint's secret part of the signing key SK_{sign}. Second, the exchanged DH values that are used for the session key computation are also signed with SK_{sign}. Hence, not only the first transmitted BARM measurement as well as the DH values are bound to the channel endpoint, but also the session key SeK that eventually establishes the secure communication channel for authentic state reporting. This means that every heartbeat message is also bound to the channel endpoint. These messages are only transmitted in encrypted form via the channel that is protected by SeK.

  The endpoint's authenticity also prevents a relay attack where the attacker could send a request to a third platform to sign platform state data PSD that includes a BARM measurement value that is less than 50 bus transactions. The third platform has no access to SK_{sign} of the target platform. That means, we can exclude that the attacker is able to conduct a relay attack. Alternatively, the attacker could try to forge a PSD signature. To do so, the attacker requires SK_{sign} that is present in the main memory. Again, when the attacker searches for SK_{sign} via DMA, BARM will reveal this attack and the memory access will be prevented. Hence, we can conclude that the attacker is unable to forge digital signatures.

- **R3 – Privacy:** The only sensitive data that is transmitted unencrypted is the first BARM measurement value that is sent to the peer during the handshake. While a compromised network interface card could be used to intercept this value, it is unlikely that this first measurement value is of use for an attacker. It is independent of further measurement values, which are required to identify when BARM determines $-\mathcal{T}$ bus transactions. Hence, we can conclude that our authentic reporting channel adheres to the least information paradigm.

6 Summary

In this technical report we developed, implemented, and evaluated an authentic reporting channel application for BARM. This channel is based on the secure channel protocol TLS. We modified the TLS protocol to consider BARM measurements during the handshake as well as during the rest of the communication session. Our modifications are based on TLS extensions. This means that our channel is compliant with the TLS specification. Furthermore, the implementation of our reporting channel fulfills the security requirements (host CPU endpoint authenticity and channel binding) that we defined for the DMA malware scenario. Without the fulfillment of these requirements malware executed on the network interface card is a threat for an authentic communication with an external platform.

Our channel is an application for our bus agent runtime monitor if platform state change reporting is required by a communication partner. The authentic reporting channel transmits the state changes to the peer. We confirmed BARM's effectiveness and efficiency with our DMA malware DAGGER in conjunction with the implemented reporting channel. Previous work that is related to authentic platform state reporting assumed the presence of an efficient runtime monitor. However, the corresponding proof of concept implementations presented in previous work did not include such a monitor. Furthermore, previous work did also not consider the DMA malware scenario.

We can also conclude that BARM can handle more complex bus masters. We demonstrated that BARM can not only handle the host CPU, the UHCI controller, etc., but also the ethernet controller. To integrate the ethernet controller into BARM's detection model we had to analyze the controller with regards to memory read and write accesses. We were able to distinguish read and write accesses by exploiting additional performance monitoring unit configurations. However, to eventually determine the number of bus transactions that are caused by the ethernet controller we had to introduce a new parameter. This new parameter is the cache line size. According to our evaluation, BARM measurement fluctuations are minimally higher as compared to the BARM version that does not consider the ethernet controller. Nonetheless, the fluctuations are still in the range of $\mathcal{T} = +/-50$ bus transactions. Our empirical measurements revealed that the performance overhead of the authentic reporting application is negligible if heartbeat messages are sent approximately every second. The reporting interval can be greater than BARM's sampling interval. The loss of DMA malware attack information is prevented by including an attack history in the heartbeat messages.

References

[1] Patrick Stewin. A Primitive for Revealing Stealthy Peripheral-based Attacks on the Computing Platform's Main Memory. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2013.

[2] Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, and N. Asokan. Beyond Secure Channels. In *Proceedings of the 2007 ACM*

Workshop on Scalable Trusted Computing, STC '07, pages 30–40, New York, NY, USA, 2007. ACM.

[3] Frederik Armknecht, Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, Gianluca Ramunno, and Davide Vernizzi. An Efficient Implementation of Trusted Channels based on OpenSSL. In *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing*, STC '08, pages 41–50, New York, NY, USA, 2008. ACM.

[4] The Computer Language Company Inc. Heartbeat. Computer Desktop Encyclopedia: `http://lookup.computerlanguage.com/host_app/search?cid=C999999&term=heartbeat&lookup.x=27&lookup.y=21` [accessed 25 February 2014], 2013.

[5] Siani Pearson, Boris Balacheff, Liqun Chen, David Plaquin, and Graeme Proudler. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002. Hewlett-Packard Professional Books.

[6] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal*, 10(3):179 –192, August 2006.

[7] Patrick Stewin and Iurii Bystrov. Understanding DMA Malware. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2012.

[8] Intel Corporation. Intel I/O Controller Hub 8/9/10 and 82566/82567/82562V Software Developer's Manual. Intel Corporation: `http://www.intel.com/content/dam/doc/manual/i-o-controller-hub-8-9-10-82566-82567-82562v-software-dev-manual.pdf` [accessed 25 February 2014], July 2009.

[9] Jon Stokes. *Inside The Machine: An Illustrated Introduction to Microprocessors and Computer Architecture*. No Starch Press Series. No Starch Press, 2007.

[10] Intel Corporation. Intel 3 Series Express Chipset Family. Intel Corporation: `http://www.intel.com/Assets/PDF/datasheet/316966.pdf` [accessed 25 February 2014], August 2007.

[11] Intel Corporation. Intel VTune Amplifier 2013 – Document Number: 326734-004. Intel Corporation: `http://software.intel.com/sites/products/documentation/doclib/iss/2013/amplifier/lin/ug_docs/index.htm` [accessed 25 February 2014], 2013. External Bus Events.

[12] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. Internet Engineering Task Force: `http://www.ietf.org/rfc/rfc5246.txt` [accessed 25 February 2014], August 2008. Network Working Group RFC 5246.

[13] Davide Vernizzi. TLS Hello Extensions and Supplemental Data. Blog: `http://tlsext-general.blogspot.de/2008/12/tls-hello-extensions-and-supplemental.html` [accessed 25 February 2014], December 2008.

[14] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. The Internet Engineering Task Force: `http://www.ietf.org/rfc/rfc4366.txt` [accessed 25 February 2014], April 2006. RFC4366.

[15] S. Santesson. TLS Handshake Message for Supplemental Data. The Internet Engineering Task Force: `http://www.ietf.org/rfc/rfc4680.txt` [accessed 25 February 2014], September 2006. RFC4680.

[16] Markulf Kohlweiss, Ueli Maurer, Cristina Onete, Björn Tackmann, and Daniele Venturi. (De-)Constructing TLS. Cryptology ePrint Archive: `http://eprint.iacr.org/2014/020.pdf` [accessed 25 February 2014], January 2014.

[17] Tilo Müller, Andreas Dewald, and Felix C. Freiling. AESSE: A Cold-boot Resistant Implementation of AES. In *Proceedings of the Third European Workshop on System Security*, EUROSEC '10, pages 42–47, New York, NY, USA, 2010. ACM.