CF-ADSI: Control Flow Anomaly Detection based upon Static Instrumentation

a dissertation presented
by
Jerome M. Peyroutat-Basse
to
The Department of Security

in partial fulfillment of the requirements
for the degree of
Engineer Degree in Computer Science
in the subject of
Security and Network

Nagoya Institute of Technology
Nagoya, Japan
October 2018

Thesis advisors: Shoichi Saito Jerome M. Peyroutat-Basse

CF-ADSI: Control Flow Anomaly Detection based upon Static Instrumentation

Abstract

Embedded device manufacturer are nowadays continuously confronted to security challenge. They will however find few security concept taking into account their strong requirement in term of performance and safety, especially in the manufacturing, healthcare, automotive and aerospace industry.
CF-ADSI attempts to provide an innovative model to detect malicious execution by aggregating the cutting edge research on embedded security, secure cloud and anomaly detection. This paper is based up the recent C-FLAT approach by Abera et al. (2016) which allows to circumvent advanced attack such the Return-Oriented Programming exploitation.
The CF-ADSI implementation reaches significant performance gain compare to C-FLAT with 70% less overhead. Moreover, this approach allows to limit the usage of security component such as HSM or TSM, thus reducing manufacturing variable cost. In return for these enhancements, the overall fixed cost of CF-ADSI is higher in order to establish the cloud and Big Data infrastructure.

Contents

To Japan and its inhabitants that inspire my everyday life.

Acknowledgments

I deeply thanks the Professor Shoichi Saito for assisting me during my stay at Nagoya Institute of Technology. I am deeply indebted for having participating to the joint meeting between the security department of Nagoya Institute of Technology and the Ritsumeikan University which was a wonderful experience.

This thesis will not have been possible without the keen help of Chikara Kato, Hiroki Ogawa, Kenta Nishimura, Kiyofumi Okano and every student in my laboratory for reaching me out when I was struggling with my Japanese. A special acknowledgement is directed toward Shohei Komatsu who actively has participated to my studies.

The last word is addressed to my family and especially to my beloved mother for offering me the opportunity to study in Japan and to fulfill my deepest dream.

0 Introduction

The security of embedded system is one of the major challenge for both manufacturer and security researcher. This is especially true for embedded system in the automotive, healthcare and IoT industry with several practical exploitation exposed over the past decade (Miller & Valasek, 2010; Kovelman, 2017; Hunt, 2016; Gayou, 2018; Frank, 2018; Zonenberg, 2016). Security in embedded system is particularly crucial because it often participate in the device's safety, and therefore to the user safety.
The technology progress in both the automotive and healthcare industry brings forward the need for security in embedded device even further. Indeed, the emergence of connected vehicule and the technology spreading in the medicine such as computer-assisted intervention enlarge the attack surface for malicious agent (Koscher et al., 2010; Checkoway et al., 2011; Rushanan et al., 2014).

Classic security solutions are however often incompatible with embedded system because of resource limitation and stronger requirement imposed by the industry. Moreover, to ensure the good implementation of security concept, the setting up of a Secure Software Development Life Cycle (S-SDLC) is generally needed (Singh & Panda, 2018). Nevertheless, the introduction of S-SDLC in organization takes time and thus future development in the next years are likely to not implement strong security concepts. How to offer an embedded system security model while limiting both the impact in resource usage and deployment time ?

This paper takes as a starting point the work of Abera et al. which elaborate a remote **C**ontrol **F**low **AT**testation (C-FLAT) model for embedded system (Abera et al., 2016). This model focuses on advanced attack taking advantage of control flow modification to execute malicious code such as the Return-Oriented Programming (ROP) attack[1].

First, we introduce the 3 major concepts used throughout our studies: control flow attestation, binary instrumentation and anomaly detection. Second, we explicit our overall design by both its underlying hypothesis and its architecture. The design is then put into practical use by implementing it in a simple environment. Finally, the result of our studies are evaluated and confronted to our design's hypothesis.

[1] `https://en.wikipedia.org/wiki/Return-oriented_programming`

1 Literature Review

1.1 Control Flow Attestation

The foundation of this paper is the **C**ontrol **F**Low **AT**testation (C-FLAT) model which enables remote attestation of an application's control flow graph (CFG) (Abera et al., 2016).

1.1.1 Design

To begin with, the controller generates the application's CFG and calculate each possible path within it. The result is stored in a measurement database in order to determine later on the correctness of the application's execution flow.

The C-FLAT model is based on a traditional remote attestation design with a nonce challenge sent by the controller and a response sent by the application (Figure 1.1). On one hand, the challenge is mainly used to prevent replay attacks. On the other hand, the response contains both the attestation report and a signature to ensure the attestation's origin. The response's signature is then verified as well as the attestation report. The application is considered as trusted if both verification are passed.

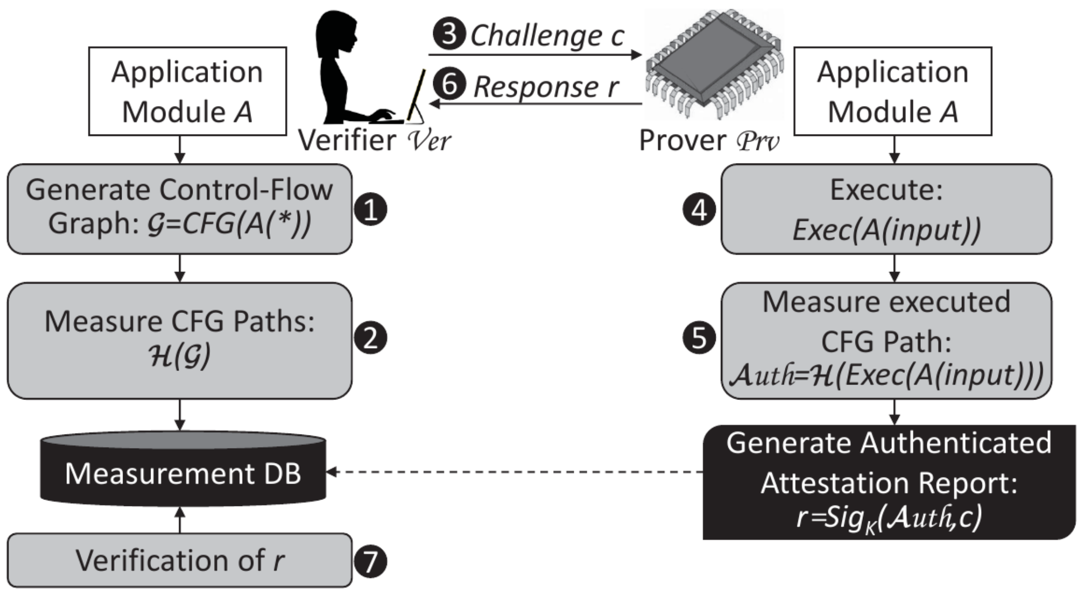


Figure 1.1: Overview of C-FLAT (Abera et al., 2016)

1.1.2 Control-Flow Attestation Report

The attestation report is generated by retrieving a basic block's identifier (e.g. starting address), say N2, when entering a basic block and add it to a hash cipher. The hash computed in the previous basic block, say H1, is also added to the hash cipher. Therefore, the attestation report becomes $H2 = Hash(N2, H1)$. The hash chain enables to identify a specific path and to validate the executed flow.
Nevertheless, the generation of the attestation report includes 3 major challenges: loops, break statements and call-return matching.

Loop: The main problem introduced by loops is that the number of valid attestation report may grows exponentially if the application has a lot of loop (or nested loop). This fundamentally affect C-FLAT's security because the number of valid attestation report must be statically low compared to the hash-space [1], otherwise the attestation report of an invalid and valid execution flow may collide, i.e. have the same hash, thus allowing an attack to occur.
In order to tackle this complication, a separate hash chain is computed when entering a loop. By doing so, the number of valid hash is limited to the number of possible path

[1]number of possible hash

in the loop. As a loop may be repeated several time, the C-FLAT model keeps track of the occurrence's number of each hash in the loop. Moreover, to ensure that the loop is executed at the right place, the hash computed right before the loop is also specified in the attestation report.
A loop is thus authenticated by 3 components: the hash computed before entering the loop, the hash computed inside the loop in a separate hash chain and the occurrence of each executed path inside the loop.

In the Figure 1.2, the *cfa_quote* represents the current executed flow. Then, the hash computed before the loop is registered as in the line 3. In the line 4, the executed path inside the loop represented by a hash and its number of occurrence (682) is saved. It is also possible to have several execution flow in a flow such as in the *loop[004]* in line 11.

```
 1 [INFO] cfa_quote: 57 92 0f 9e 98 47 30 bb a5 f7 5d 2a dc 8a 7b 5f
 2
 3 [INFO] loop[000]: b3 c5 ca c4 6f dc 6a d0 4a 80 10 09 af a3 59 70
 4 [INFO] path[000]: 97 78 fb fc 93 09 4e d7 ac 32 5d 65 eb 29 08 0c (682)
 5 [INFO] loop[001]: eb 16 88 8a d2 3b c6 19 f9 01 94 5d ee cb 1c 13
 6 [INFO] path[000]: b9 7d cf 8d 00 b6 5f 63 b3 7c 60 e4 e3 be 56 17 (1)
 7 [INFO] loop[002]: 6d 05 6e b2 3a 27 1e 2b 78 3e f9 4c e3 a7 cb f8
 8 [INFO] path[000]: 62 f7 b8 0b 65 4b de 35 c7 05 bc 28 06 43 11 6e (2)
 9 [INFO] loop[003]: eb 16 88 8a d2 3b c6 19 f9 01 94 5d ee cb 1c 13
10 [INFO] path[000]: b9 7d cf 8d 00 b6 5f 63 b3 7c 60 e4 e3 be 56 17 (3)
11 [INFO] loop[004]: f5 77 b7 94 bd 6c 81 e2 2f 36 da ad cd df 56 6e
12 [INFO] path[000]: 67 c6 5e d4 18 13 02 bc 4a 5d 60 a0 16 85 f4 ed (9)
13 [INFO] path[001]: 78 19 af 09 0f d5 64 f4 39 b4 7a 0d 97 57 77 8c (2)
```

Figure 1.2: Attestation Report (Abera et al., 2016)

Break statements: Break statements must be considered as special loop exit. Indeed, they directly leave the loop without going back to the conditional check. The hash chain in the loop is thus extended to the targeted exit address. Therefore, the final hash capture the fact that the loop terminated by a break statement toward a specific exit address.

Call-return matching: Call and return instructions become a challenge when a function is called from different places (N2, N3) in the application as in Figure 1.3. In this case, the function may return to N5 or N6. However, if the path calculation is not done carefully, the path N2->N4->N6 may be seen as valid and thus lead to an attack.
In the C-FLAT model, an index is added to the call and return edge to identify which return correspond to which call.

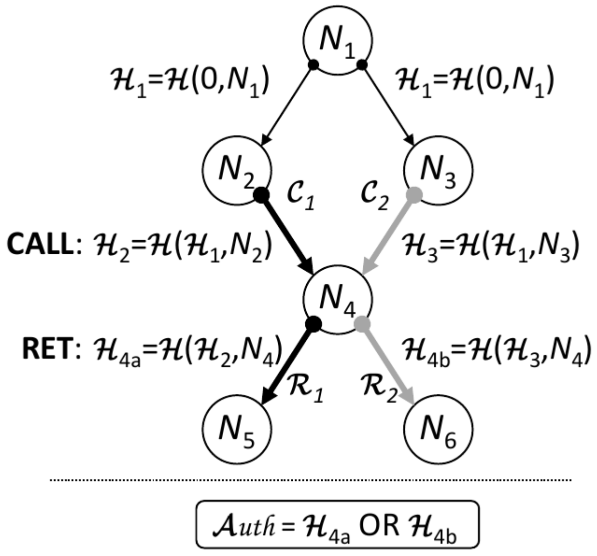

N_1
$\mathcal{H}_1=\mathcal{H}(0,N_1)$
$\mathcal{H}_1=\mathcal{H}(0,N_1)$
N_2
N_3
$\mathcal{C}_1$
$\mathcal{C}_2$
CALL: $\mathcal{H}_2=\mathcal{H}(\mathcal{H}_1,N_2)$
$\mathcal{H}_3=\mathcal{H}(\mathcal{H}_1,N_3)$
N_4
RET: $\mathcal{H}_{4a}=\mathcal{H}(\mathcal{H}_2,N_4)$
$\mathcal{H}_{4b}=\mathcal{H}(\mathcal{H}_3,N_4)$
$\mathcal{R}_1$
$\mathcal{R}_2$
N_5
N_6
$\mathcal{A}uth = \mathcal{H}_{4a}$ OR $\mathcal{H}_{4b}$

Figure 1.3: Overview of C-FLAT (Abera et al., 2016)

1.2 Binary Instrumentation

The implementation of a control flow attestation requires to retrieve basic block's identifiers. Binary instrumentation methods allow us to arbitrary insert assembly code in the application's binary. In the following section, we introduce two major concepts used to instrument an application: Dynamic and Static Binary Instrumentation.

1.2.1 Dynamic Binary Instrumentation

The dynamic binary instrumentation analyses and executes callback routines as the application is running. The instrumentation tool detects basic block and event to be instrumented by monitoring the executed application. Developers define which condition will trigger callback routines. If a basic block or a event passes the condition, an user defined callback routine is called and the application is resumed once the callback routine is finished. Most common toolkit to implement dynamic instrumentation are Intel PIN[2], DynamoRIO[3], Dyninst[4] and Valgrind[5].

[2] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
[3] http://www.dynamorio.org/
[4] https://www.dyninst.org/
[5] http://valgrind.org/

Pros

- Do not modify the targeted application
- Retrieve detailed information of the executed application

Cons

- Cannot instrument statically compiled language
- Have a significantly overhead both in time and memory because of the external process attached to the application

1.2.2 Static Binary Instrumentation

The static binary instrumentation encompasses 2 approaches: the pre-compilation and the post-compilation method. On one hand, the pre-compilation method analyses the source code of the application and insert code blocks into the region to be instrumented. For example, the instrumentation may be done by the compiler. On the other hand, the post-compilation method analyses the compiled binary to find every methods. Then, assembly blocks are inserted and the binary is rewritten. Most available toolkit are issued from research such as PIN, UROBOROS and PEBIL (Luk et al., 2005; Wang et al., 2016; Laurenzano et al., 2010). The toolkit Dyninst[6] also offers static instrumentation.

Pros

- Have a relatively small overhead compare to dynamic instrumentation

Cons

- Significantly modify the application's binary
- Binary rewriting is arduous to implement
- Lack of detailed analysis such as run-time information

1.3 Anomaly Detection

The generation of the CFG and its analysis is one of the most challenging aspect in the C-FLAT model. Nevertheless, anomaly detection methods may provide us solutions to

[6]`https://www.dyninst.org/`

avoid the CFG's generation and its analysis. To investigate such paradigm, we consider each attestation report generated as a log information, and thus direct our attention toward anomaly detection in log file.

1.3.1 Supervised and Unsupervised Anomaly Detection

The 2 major theories in anomaly detection are supervised and unsupervised detection (Omar et al., 2013).
A supervised detection requires data labeled as "normal" or "abnormal" in order to train a classifier. The classifier is then used to evaluate whether a new input is unusual or not. Data used to build the classifier are thus extremely decisive as they will influence the anomaly detection system's reliability.
Contrarily, an unsupervised detection do not use labeled data and instead presumes that most of values are normal. The detection is done by identifying data that seems to not fit the remainder of the data set.
These 2 techniques can also be combined to form a semi-supervised anomaly detection. This method needs a "normal" training data set in order to construct a model representing the "usual behavior". Later on, the constructed model can modify itself by using, for example, neuro-evolution and deep learning concepts (Arunraj et al., 2017).

1.3.2 Big Data Framework

Anomaly detection needs wide data set in order to be efficient. As one of our hypothesis is that our system will handle a large number of device, we will be able to collect enough data. However, this implies to have the required infrastructure to handle such extensive amount of data. This is why different Big Data frameworks are considered in order to design our system.

Although the research on Big Data is vivid, current Big Data solutions are still not mature. Each Big Data solution has a different approach for specific goals such as real-time analysis, simplicity of use, scalability or compatibility with existing system.
In our research's context, most important criteria are the setup's ease and the real-time performance. On one hand, the setup's ease ensure us to deliver a complete system although lacking of experience in the Big Data field. On the other hand, the focus on real-time performance ensure us to handle fast streams of incoming data.
To meet real-time expectation, NoSQL[7] database solution such as MongoDb, HBase or Cassandra are preferred for data storage (S & Mary, 2017), and Spark or Storm are envisaged as analytic engine (Inoubli et al., 2018). Our investigation lead us to a

[7]also called "non relational" database

complete and comprehensive guide to setup MongoDb with Spark that settle our choice on MongoDb and Spark (Kalan, 2015).
The reader may thus recall that the choice of the Big Data Framework is based mainly on the setup's ease over performance and scalability.

1.3.3 Log Data Collection

The last step in order to design our log file anomaly detection is to collect data from log file. Several solutions are envisaged such as Fluentd, Syslog-ng, Rsyslog and Nxlog (Vaarandi & Niziński, 2013). Fluentd[8] is finally preferred because of available plugins for BigData, the possibility to develop custom plugin in Ruby and the compatibility with MongoDb.

[8]https://www.fluentd.org/

2
System Architecture

2.1 Design

The overall design of our solution is first introduced before entering into details in the system architecture. The exposed design will help to have a better understanding of our model on a functional point of view.

2.1.1 Hypothesis

To begin with, it is crucial to explicitly formulate the hypothesis upon our system is designed.

The first and the most critical assumption is that the device must has continuous access to Internet. This hypothesis seems realistic for embedded system in the healthcare or the manufacturing sector. It is however more difficult to achieve in automotive or aerospace industry. However, the expansion of technology such as LTE may allow these industries to meet this requirement.

Hypothesis 2.1.1.1. *The device has continuous access to the Internet.*

While designing our system, we presume that embedded manufacturers try to reduce the device manufacturing's variable cost to benefit from economy of scale. Therefore, our model tries to minimize the use of components such as secure memory, TPM or HSM. Moreover, the instrumentation performance on the device is also taken into account because a lost in performance may need to be compensate by a more efficient processor or microcontroller.

Hypothesis 2.1.1.2. *One of the major priority of embedded manufacturers is to reduce the variable cost of embedded device to leverage economy of scale.*

The generation of the CFG and its analyze are also considered as a obstacle that may prevent industry to consider our system. The C-FLAT model was implemented and tested on relatively simple application therefore this aspect was not taken into account.
First, the CFG's generation may be problematic because of specific instruction set architecture (TriCore, MSP, PowerPC, ARM etc...). Moreover, feature such as function pointer is also a challenge for establishing the CFG (Xu et al., 2010; Flake, 2002).
Second, if the CFG to be attested is too large, it may be impossible to calculate every path in it as the complexity of such computation is NP-Complete (Thorelli, 1966; Simões, 2009).
Finally, the measurement of the CFG must exactly match the measurement done in the device, i.e. each basic block's labeling must the same in the device and in the measurement engine of the attestation server.

Hypothesis 2.1.1.3. *The generation of the CFG is not viable when considering the system's deployment in the industry.*

We also assume that such advanced attestation model will be adopted in the case where the embedded device's safety is a priority. This may be for example a device monitoring the insulin level, a manufacturing machine, a radiography device or an actuator in a vehicle. The second priority after safety is often the device's performance. As an instance, a loss in performance may induce loss in productivity for a manufacturing machine or a delay in response time for an actuator.
Some IoT device focused on security such as smart lock may also benefit from our approach. However these tyoe of devices are still marginal and do not arise a strong business matter such as in the automotive, healthcare or manufacturing field.

Hypothesis 2.1.1.4. *The device security is often a concern among embedded device that needs high level of safety. The security design must although limits its impact of the device performance.*

2.1.2 System Overview

The **C**ontrol **F**low **A**nomaly **D**etection based upon **S**tatic **I**nstrumentation (CF-ADSI) system is developed with the following goals:

- Detect anomaly in the binary execution flow
- Have a minimal impact on the device performance
- Do not depend on the control flow graph

These requirements lead to 2 main concepts. On one hand, the hash computation is exported to a trusted cloud as it represents 80% of the C-FLAT overhead (Abera et al., 2016). By trusted cloud, we implies that the hash computation or its results cannot be tampered by the cloud provider environment (another application, operating system etc...). On the other hand, anomaly detection in the execution flow will be preformed by Big Data analysis framework on resulting CFG attestation.

The system design is composed of 3 components: devices, servers in a trusted cloud and a Big Data analytic framework (Figure 2.1).

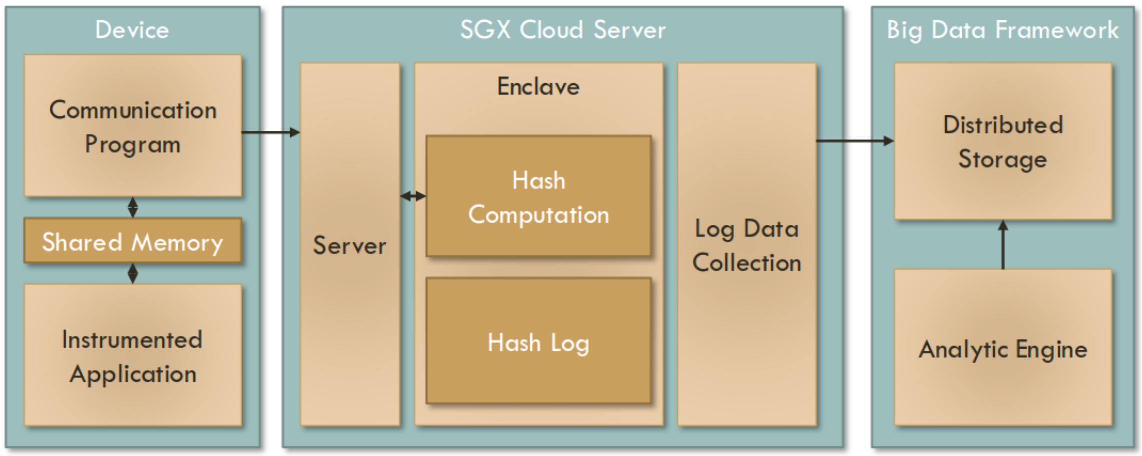


Figure 2.1: CF-ADSI Overall Design

Device

The device is divided in 2 parts: the instrumented binary and a communication program. This division is not fundamentally necessary, but this allows us to easily switch from one instrumentation toolkit to another if needed.

The binary is instrumented in order to retrieve basic block identifier which can be the basic block's starting address or an arbitrary identifier defined during the instrumentation. The basic block identifier is then placed into a shared memory space. The communication program reads each identifier inserted into the shared memory and forward it to the SGX cloud server.

The instrumented binary and the communication program must also be synchronized by mutex, lockfree queue etc. This adds a significant overhead that will be measured at a late stage in this paper. Nevertheless, the communication stack can also be implemented during the binary instrumentation, thus eliminating overhead related to the I/O synchronization in shared memory.

SGX Cloud Server

The server in the SGX cloud make the transition between the device and the Big Data framework as well as doing the hash computation. The SGX enabled cloud allows to ensure that the hash calculation is not tempered by creating enclaves. Enclave have their own execution space protected at the CPU level and specific instructions are implemented in the processor to communicate with enclave. Therefore, it is not possible to temper or access information belonging to an enclave even if the operating system of the cloud provider is corrupted [1].

Basic block identifiers are received from the device and then passed to an enclave by using the SGX library API. The enclave compute a new hash with the basic block identifiers and update the CFG attestation. This report is then logged in the operating file system. A log collection deamon is used in every SGX cloud server to gather logs in our Big Data framework.

Big Data Framework

Logs are then analyzed in a Big Data framework composed of a distributed database or storage and an Big Data analytic engine. As presented in the literature review, there is a vast choice of solution to build such a system. Our Big Data framework is using MongoDB as a distributed database and Apache Spark as an analytic engine as stated in the literature review (see section 1.3.2).

2.2 Device Application

The section details the implementation phase of the device in the CF-ADSI model.

[1]`https://software.intel.com/sgx`

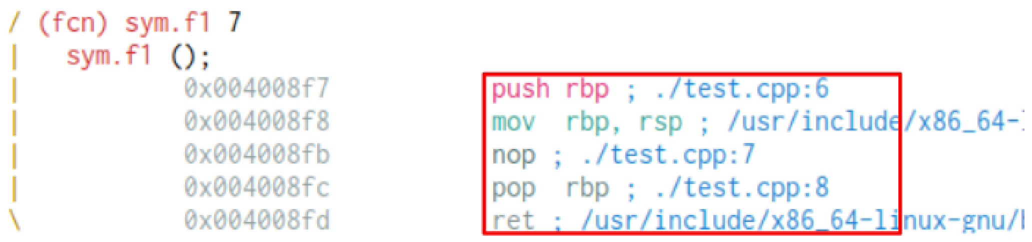
2.2.1 PEBIL Instrumentation

According to our design, the static instrumentation methods is chosen to limit performance overhead in the device. The PEBIL toolkit offered by Laurenzano et al. is prefered because of the small overhead and the possibility to easily modify the toolkit if needed (Laurenzano et al., 2010).

The static instrumentation implemented in the PEBIL toolkit follows the same framework as the DynInst`https://www.dyninst.org/` toolkit. First, the control flow graph is analyzed to identify every methods in the binary. Then, the beginning of each methods is rewritten with a jump call to a new section. Indeed, one of the biggest challenge in static instrumentation is to have the required space to insert assembly code. Therefore, the entire method is rewritten into a new section[2] to insert assembly code of arbitrary length.
Classic static instrumentation toolkit such as DynInst allows only to insert instrumentation function in the binary. This is often done by adding a call instruction during the method's rewriting. However, this method induce a stronger overhead because of the context switch overhead, i.e. the protection of memory stack and CPU's registers. Nonetheless, the PEBIL toolkit offers us to write custom assembly codes and insert them directly during the method's rewriting, thus limiting the overhead related to context switch.

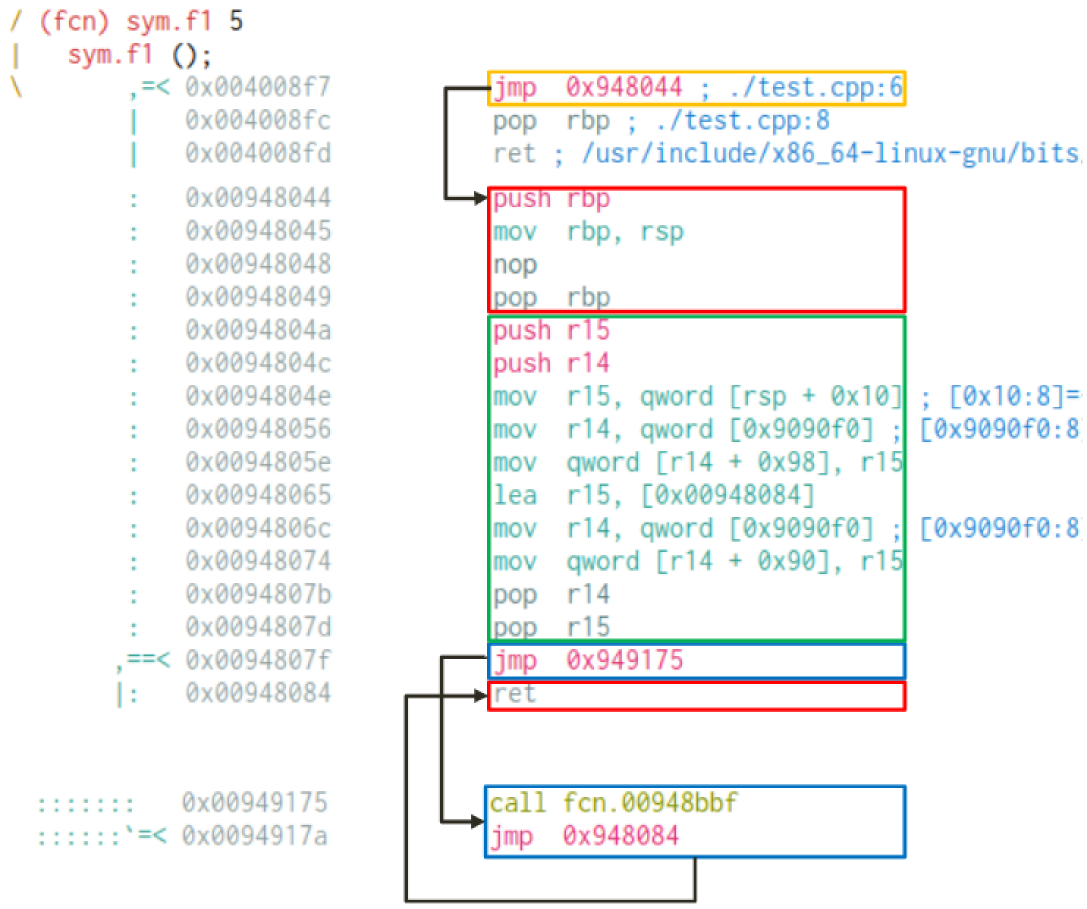
In Figure 2.2 and 2.3, a simple example is presented to give more substance to our explanation. The assembly code of the function named ”f1” is straightforward. The function ”f1” starts at the address 0x004008f7 and does nothing as indicated by the ”nop” instruction. During the instrumentation process, the beginning of the ”f1” function is first rewritten with a jump instruction (in yellow) to a new section. Then the ”f1” function's assembly code is copied at the address 0x948044 into the new section (in red). This allows use to insert assembly code just before the ”ret” instruction. From address 0x0094804a to 0x0094807d (in green), we use PEBIL to directly insert custom assembly code without using an instrumentation function. However, at address 0x0094807f (in blue), we use an instrumentation function. Instrumentation function is constituted to a first jump to the address 0x00949175 (in blue), and then a library function that we have defined is called. Finally, we return to the ”f1” function by a jump to the address 0x00948084 (in red).

[2]refers to binary section such as .txt, .data or .rodata



```
/ (fcn) sym.f1 7
|   sym.f1 ();
|            0x004008f7      push rbp ; ./test.cpp:6
|            0x004008f8      mov  rbp, rsp ; /usr/include/x86_64-
|            0x004008fb      nop ; ./test.cpp:7
|            0x004008fc      pop  rbp ; ./test.cpp:8
\            0x004008fd      ret ; /usr/include/x86_64-linux-gnu/l
```

Figure 2.2: Assembly code before instrumentation



```
/ (fcn) sym.f1 5
|   sym.f1 ();
\       ,=< 0x004008f7      jmp  0x948044 ; ./test.cpp:6
        |   0x004008fc      pop  rbp ; ./test.cpp:8
        |   0x004008fd      ret ; /usr/include/x86_64-linux-gnu/bits
        :   0x00948044      push rbp
        :   0x00948045      mov  rbp, rsp
        :   0x00948048      nop
        :   0x00948049      pop  rbp
        :   0x0094804a      push r15
        :   0x0094804c      push r14
        :   0x0094804e      mov  r15, qword [rsp + 0x10] ; [0x10:8]=
        :   0x00948056      mov  r14, qword [0x9090f0] ; [0x9090f0:8
        :   0x0094805e      mov  qword [r14 + 0x98], r15
        :   0x00948065      lea  r15, [0x00948084]
        :   0x0094806c      mov  r14, qword [0x9090f0] ; [0x9090f0:8
        :   0x00948074      mov  qword [r14 + 0x90], r15
        :   0x0094807b      pop  r14
        :   0x0094807d      pop  r15
      ,==< 0x0094807f      jmp  0x949175
      |:   0x00948084      ret

 :::::::    0x00949175      call fcn.00948bbf
 ::::::`=< 0x0094917a      jmp  0x948084
```

Figure 2.3: Assembly code after instrumentation with the PEBIL toolkit

The PEBIL toolkit is used to achieve the following instrumentation:

- Retrieve basic block starting address
- Write retrieved addresses into the shared memory

- Detect the start and end of a loop

The collection of basic block starting address and the loop detection are done with custom assembly code directly injected into the binary. However, an instrumentation function is used to write into the shared memory because we used a loockfree queue from the boost library. It is however possible but more complicated to use the boost API directly by inserting custom assembly code.

2.2.2 Shared Memory and Communication Program

The shared memory space is set up by the instrumented device in instrumentation function at start up. The boost library API is used to create and manage the shared memory. The synchronization between the instrumented application and the communication program is done by using a lockfree queue from the boost library. The communication program retrieves the basic block starting address in the queue and creates a new Google Protobuf[3] message to be send to the SGX cloud server. The queue's fetching and the message sending is done in separate thread to limit overhead.

2.3 SGX Cloud Server

The server's implementation is focused on its functionality more than its security. The hash calculation is run in an SGX enclave but security on other part is overlooked. For example, the network stack or the logging function may be vulnerable.

2.3.1 Untrusted Application

The untrusted application is in charge of receiving basic block starting address from devices. Devices are given a distinct ID in order to compute a distinct CFG attestation for each device. Basic block starting addresses are received in a Google Protobuf[4] message and then passed to the enclave through the Intel SGX API[5].
The Intel SGX library does not provide I/O library for enclaves. Therefore, CFG attestation is also written in the log file by the untrusted application.

2.3.2 Enclave

The enclave retrieves basic block starting address from through the SGX API. It then computes the new hash given the current hash and the retrieved address. The Intel

[3]https://developers.google.com/protocol-buffers/
[4]https://developers.google.com/protocol-buffers/
[5]https://software.intel.com/en-us/sgx-sdk-dev-reference

SGX Integrated Performance Primitive Library[6] (IPP) is used to compute the hash as efficiently as possible. The new CFG attestation is then generated and sent to the untrusted application to be written in the log file.
As in the C-FLAT model, a new hash instance is created when a loop is entered. However, contrary to the C-FLAT model, only the current hash[7] is logged to be analyzed by the Big Data framework. The complete CFG attestation (see Figure 1.2) can also be logged to perform a more detailed analysis. Along side to the current hash, the current time and the associated device id is logged.

2.3.3 Log File Collection

A fluentd[8] deamon is executed on each server to gather information from the log file to our distributed database. A custom parsing plugin is developed to parse the log file. The current time, the device id and the current hash are thus sent to the distributed MongoDB database.

2.4 Big Data Framework

The implemented Big Data framework is relatively simple and is divided into 2 components: a distributed database and an Big Data analytic engine.

2.4.1 Distributed Database

As explained in the literature review and the design, MongoDB is chosen to perform real-time analysis. MongoDB is a NoSQL database, i.e. it doesn't use SQL[9] and do not need to have predefined layout such as table. Instead, MongoDB store data as BSON[10] document. If several BSON document has the same structure, i.e. same field, then they are organized as collection. Therefore, the data structure is unknown until entries are added to the database. Figure 2.4 gives an example of an entry in the MongoDb database.

```
> db.hash.findOne()
{
  "_id" : ObjectId("5b949dc55ff9874fe00932cf"),
  "id" : "2",
```

[6]https://software.intel.com/en-us/ipp-crypto-reference
[7]corresponding to the *cfa_quote* in Figure 1.2
[8]https://www.fluentd.org/
[9]strutured query language
[10]binary JSON

```
  "hash" : "efe3f94f2777c7b6c535eb3bcc39ac4ecdf782f81b4b0124a253fe851eee2ab2",
  "time" : ISODate("2018-09-09T04:12:43Z")
}
```

Figure 2.4: MongoDb Entry in the CF-ADSI model

2.4.2 Analytic Engine

The analytic engine in our implementation is Apache Spark[11]. It easily integrates with MongoDB and has a Python2 and Python3 API named PySpark. Python is favored because it is one of the most common language in the Big Data field with Scala and allows to quickly elaborate prototype.

The anomaly detection algorithm used is rather simple:

- Calculate the occurrence frequency of each hash in the MongoDB database
- Detect a hash as unusual if the occurrence frequency falls under a arbitrary threshold

For example, in the Figure 2.6 the hash '29 a9 45 c7 d4 9c 6f c4 2c a1 b8 9b 65 b1 71 21 e2 19 e0 6b 01 f6 a7 98 e5 d2 24 ff 3b 59 bb 8f' as the lowest occurrence frequency. If this frequency fall behind the threshold, then the device associated with this hash is considered to have an unusual execution flow. This can be due to an attack such as a ROP or a due to a bug. This unusual execution flow can also be an extremely rare legit execution flow, such as a function that is almost never used. Therefore, further investigation is needed to determine the anomaly's cause.

```
Row(hash='06498546305fd6077c1d293f911e1c092f320b7fbe8deaf14a85397ca72da00f', count=1114)
Row(hash='c26f95bbe71d097e592892901bac8dd2c266b0a003c173ef7fff716469c281a3', count=101)
Row(hash='f10732db93211adce1cf56560ca28ba751aba3ec29887f1c80429d432fe7bc97', count=1114)
Row(hash='4a74c9e514229bf2c272b2fa5d5c1151469c9d0ec9d2e9175dc59b37bee2dbe4', count=100)
Row(hash='8935c115c8b9fabcba28c31da8d4a2d6691152ff416ad9ad2ad7ccdedd389fd6', count=1114)
Row(hash='2fb1c65bd38fdf901675b4f5cd744305ad3821f92f9f4d573bc1807f03b40eea', count=100)
Row(hash='1a561e9c398a2bd323b26dbc85345ee131fffdc9aa0806601b65db6197ad4809', count=1215)
Row(hash='7fcaf9a1e8339469ab3e20b5579cff714644a9f72fc099efec98cd22bbce451b', count=100)
Row(hash='c14bd28cc2c4a81beca82905c47a9d8a3f844bb060c12dac12f4c7e596842a8b', count=100)
Row(hash='4653e23a86c04491bf105b1af6926afa68f838f22f48981483ccff19c32470d4', count=100)
Row(hash='efe3f94f2777c7b6c535eb3bcc39ac4ecdf782f81b4b0124a253fe851eee2ab2', count=101)
Row(hash='de5b6959ea27850fa396a4b6f7bbc6022c95e5a7abc3e1b33091c472abad0d8e', count=101)
Row(hash='32844033257d93f3a43bc1248bd1bdb681ef5cfcb2e493e2808fc0a0595ed133', count=101)
Row(hash='1a41260da07beeb370afb2a6383bead9f8ca8edc0e8cb785a06c6c1d0e257c69', count=101)
Row(hash='6253af85d391043ae73855ab2271dd955a72eebb3c15a0a0f61f8f805b891817', count=100)
Row(hash='b4dfe037d5b71d71a5077837acda6cb2bd91a205ad712c7ca3b8dfd1751e13d8', count=101)
Row(hash='78ff36bd0cf957dc35a82eff5982add5e9d28312e87f0bb7bbbcf9e4df8d33a5', count=101)
Row(hash='1fb4ad0eb72499930bfceba91640c42e485b6cb7b4ec45d9fd7768216f6683fc', count=101)
Row(hash='6963319dd9cdcca8b6770d6da1b349f3f3c604e042481fba0724adc2ca6e5848', count=100)
```

[11] http://spark.apache.org/

```
Row(hash='2add7da9d4c6ea092d9f6273f874657a43f6bf9fd62af8d26450fe9f594c2879', count=1114)
Row(hash='29a945c7d49c6fc42ca1b89b65b17121e219e06b01f6a798e5d224ff3b59bb8f', count=1)
```

Figure 2.5: Number of occurrence for each distinct hash retrieved by a GroupBy aggregation in Apache Spark

```
('29a945c7d49c6fc42ca1b89b65b17121e219e06b01f6a798e5d224ff3b59bb8f', 0.0001392757660167131)
('4a74c9e514229bf2c272b2fa5d5c1151469c9d0ec9d2e9175dc59b37bee2dbe4', 0.013927576601671309)
('6963319dd9cdcca8b6770d6da1b349f3f3c604e042481fba0724adc2ca6e5848', 0.013927576601671309)
('2fb1c65bd38fdf901675b4f5cd744305ad3821f92f9f4d573bc1807f03b40eea', 0.013927576601671309)
('7fcaf9a1e8339469ab3e20b5579cff714644a9f72fc099efec98cd22bbce451b', 0.013927576601671309)
('c14bd28cc2c4a81beca82905c47a9d8a3f844bb060c12dac12f4c7e596842a8b', 0.013927576601671309)
('6253af85d391043ae73855ab2271dd955a72eebb3c15a0a0f61f8f805b891817', 0.013927576601671309)
('4653e23a86c04491bf105b1af6926afa68f838f22f48981483ccff19c32470d4', 0.013927576601671309)
('de5b6959ea27850fa396a4b6f7bbc6022c95e5a7abc3e1b33091c472abad0d8e', 0.014066852367688022)
('32844033257d93f3a43bc1248bd1bdb681ef5cfcb2e493e2808fc0a0595ed133', 0.014066852367688022)
('1a41260da07beeb370afb2a6383bead9f8ca8edc0e8cb785a06c6c1d0e257c69', 0.014066852367688022)
('1fb4ad0eb72499930bfceba91640c42e485b6cb7b4ec45d9fd7768216f6683fc', 0.014066852367688022)
('efe3f94f2777c7b6c535eb3bcc39ac4ecdf782f81b4b0124a253fe851eee2ab2', 0.014066852367688022)
('b4dfe037d5b71d71a5077837acda6cb2bd91a205ad712c7ca3b8dfd1751e13d8', 0.014066852367688022)
('c26f95bbe71d097e592892901bac8dd2c266b0a003c173ef7fff716469c281a3', 0.014066852367688022)
('78ff36bd0cf957dc35a82eff5982add5e9d28312e87f0bb7bbbcf9e4df8d33a5', 0.014066852367688022)
('8935c115c8b9fabcba28c31da8d4a2d6691152ff416ad9ad2ad7ccdedd389fd6', 0.1551532033426184)
('f10732db93211adce1cf56560ca28ba751aba3ec29887f1c80429d432fe7bc97', 0.1551532033426184)
('2add7da9d4c6ea092d9f6273f874657a43f6bf9fd62af8d26450fe9f594c2879', 0.1551532033426184)
('06498546305fd6077c1d293f911e1c092f320b7fbe8deaf14a85397ca72da00f', 0.1551532033426184)
('1a561e9c398a2bd323b26dbc85345ee131fffdc9aa0806601b65db6197ad4809', 0.1692200557103064)
```

Figure 2.6: Frequency of occurrence for each distinct hash

3 Evaluation

Finally, this section details the performance's impact of the CF-ADSI model on the device. The Big Data analytic algorithm performance will also be analyzed to evaluate the viability of our log anomaly detection on larger data set. The hardware and software environment used in the evaluation are specified in Table B.1 and B.2 of the Appendix B.

3.1 Device Performance

The hash calculation time and the instrumentation overhead is first needed to compare the performance of CF-ADSI model to the C-FLAT model. Each time will be expressed as "time by basic block" to limit bias on measured time. The measurement below 1 million basic block are discarded because the impact of the initialization function (shared memory etc...) is too significant.

As seen in the Figure 3.1, the overhead for a basic block in the CF-ADSI model is stabilizing between 1.20 and 1.30us. The hash calculation time on the SGX cloud server is however stabilizing at 4.65us by basic block. This means that we have actually saved 4.65us for each basic block by exporting the hash calculation to the cloud. We supposed that the hash calculation on the application and in the SGX enclave is the same as the CPU performance are very close on both environment. The CF-ADSI model has thus approximately 70% less overhead on the device than the C-FLAT.

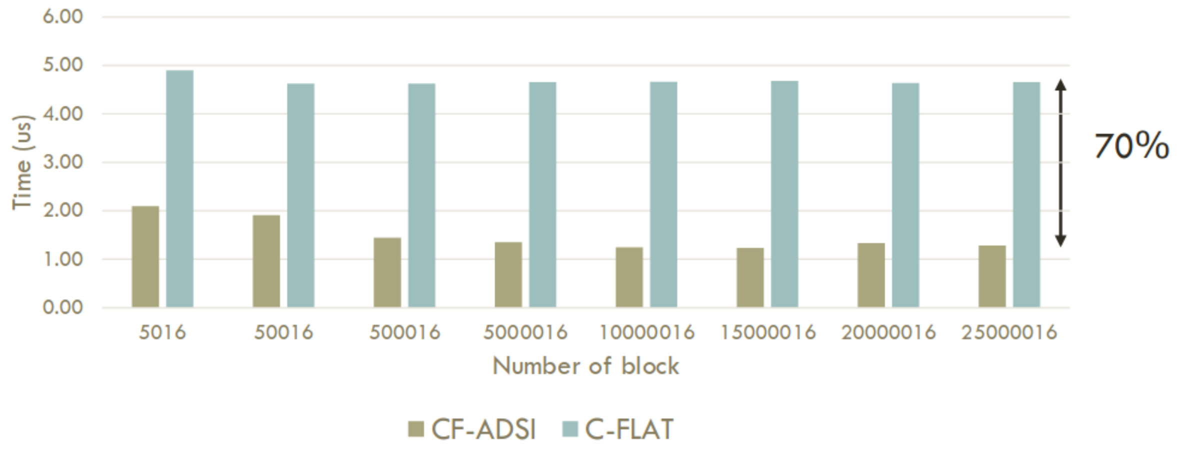


Figure 3.1: Overhead by basic block in the device application

Although this performance gain, the CF-ADSI instrumentation still induces an overhead into the device. The CF-ADSI overhead is composed by: 5% of context switch and custom assembly instruction, 45% of I/O and synchronization in the lockfree boost queue and 50% to create and send the Google Protobuf message to the SGX cloud server (Figure 3.2).

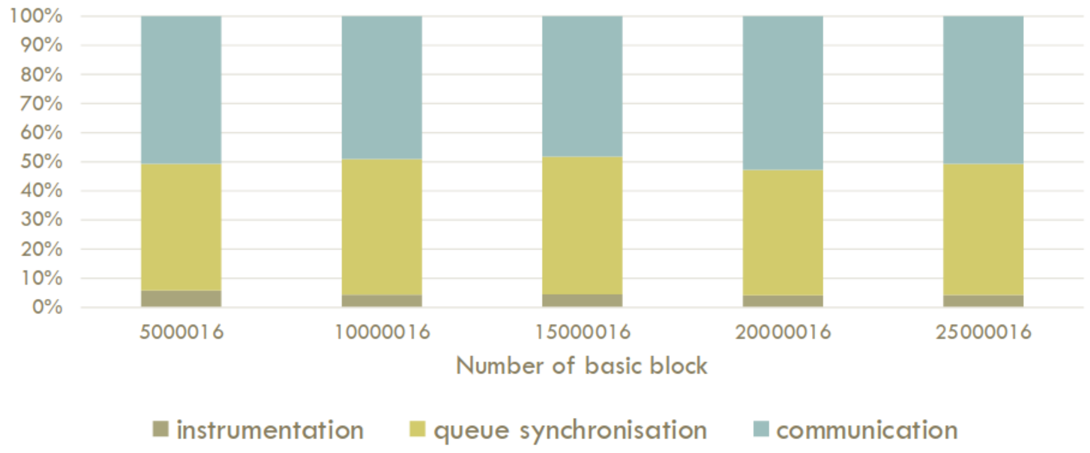


Figure 3.2: Composition of the overhead in the device

The overhead of CF-ADSI on a simple application (see Annexe A) is around 1000% to 1500% (Figure 3.3). This overhead is however related to the basic block density in the binary. For this application, the average time to execute a basic block is between 1 and 1.5ns without instrumentation. The overhead expressed in percentage is thus highly dependent on the application.

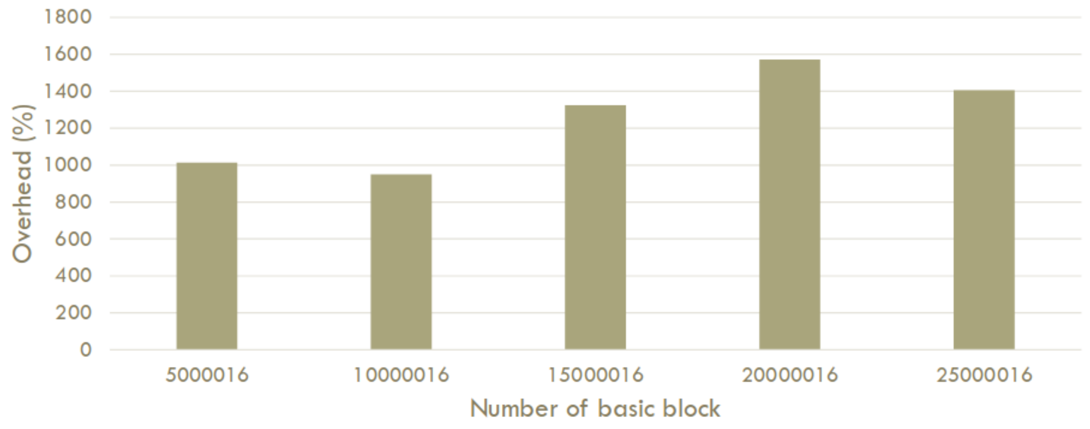


Figure 3.3: Overhead in the device application (in %)

3.2 Analytic Engine Performance

The performance of the analytic algorithm is measured to verify if such system can scale for billion or trillion of device. As shown in the Figure 3.2, the analyze time is linear and should thus be scalable.
The algorithm presented in this paper is however extremely simple and the analyze complexity may change according to the algorithm or the Big Data framework used. Further studies are definitely needed to confirm and/or improve the CF-ADSI from the anomaly detection perspective.

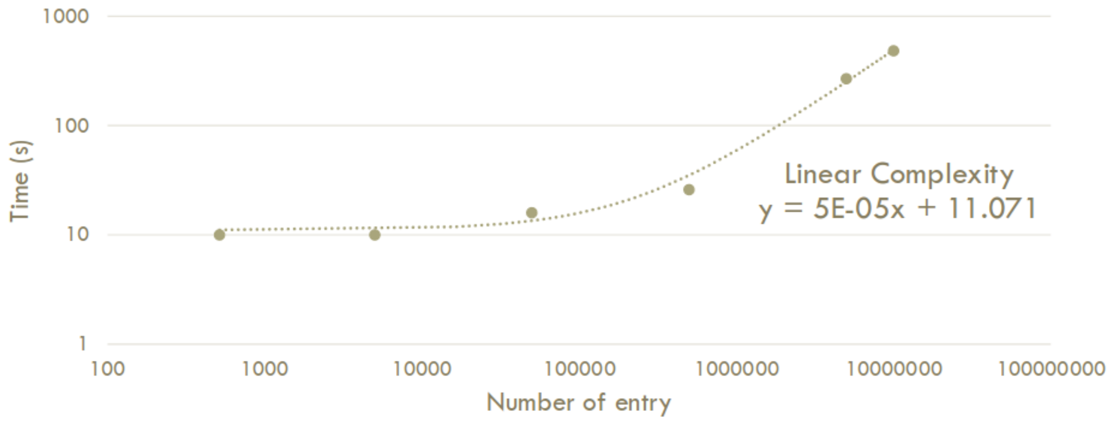


Figure 3.4: Execution time of the analytic engine algorithm with Apache Spark

4 Conclusion

This paper brought together the control flow attestation, the SGX technology and the log anomaly detection to design the CF-ADSI model. The CF-ADSI model is focused on the device performance as well as reducing resource usage such as memory on the embedded device. Moreover, CF-ADSI tackles a major limitation of the C-FLAT by eliminating the analysis of the Control Flow Graph.

The design is divided into 3 components: instrumented devices, SGX servers and a Big Data framework. The application's instrumentation is done thanks to the PEBIL toolkit. This toolkit offers a great flexibility by providing both instrumentation function and tools to directly insert custom assembly code into the application's binary. The SGX server takes advantage of the SGX platform to securely perform the hash computation inside enclaves. Therefore, CF-ADSI provides the same guarantees than using a TPM or HSM module in the device. The real-time performance and the set-up easiness had driven us to a Big Data framework founded upon MongoDB, a distributed database, and Spark, an analytic engine.

The CF-ADSI evaluation shows up a dramatic gain in performance compare to the C-FLAT model with approximately 70% less overhead. Nevertheless, the overhead is still significant with an average overhead of 1 to 1.5us by basic block. The overhead impact on a real-time embedded device is strongly dependent on the instrumented function or application. Therefore, it is difficult to assert if the CF-ADSI model can be deployed in a production environment.
However, the implemented analytic algorithm on our Big Data framework displays a linear complexity. Although simplistic, this algorithm should scales well with a larger number of device.

Nonetheless, our measurement was restrained to our testing environment and the CF-ADSI was not confronted to a strong real-time constraint, therefore limiting us to assess the viability of CF-ADSI for a real-time embedded system.
Moreover, even if the needed CFG during the instrumentation do not have to be as detailed as in the C-FLAT model, its generation may be a major limitation for uncommon instruction set architectures.
The PEBIL toolkit is also limiting the scope of our implementation by supporting only x86 architecture. Furthermore, PEBIL cannot instrument statically compiled executable and therefore library call such as system call was not taken into account in the CFG attestation.
Moreover, our implementation is far to be perfect on a security level. As an example, the communication between the device and the SGX server was done in clear without using SSL[1]. Thus, the CF-ADSI may include additional overhead in order to secure the overall system.
The chosen Big Data framework may also not be the most adapted to implement log anomaly detection. Other solutions may strongly improve performance and be more suitable. Besides, the analytic algorithm used is extremely straightforward and is surely not the most pertinent algorithm to perform the control flow anomaly detection.

This paper may be easily enlarged from the security to the passive monitoring field. Indeed, other information may be collected using our method although our research focused on retrieving information about the execution path. The CF-ADSI model can thus be widen to a more general **A**nomaly **D**etection based upon **S**tatic **I**nstrumentation (ADSI) concept.
Scholars may also be interested to try other instrumentation methods in order to improve the device performance. Our design is especially useful in this case as the instrumented binary and the communication layer are separated, thus limiting the code to be rewritten by switching the instrumentation toolkit.
The impact of implementing standard security practices on the overall CF-ADSI system may also be detailed. This will definitively help the industry to perceive this system as a potential business solution.
Researchers in the Big Data field can as well provide their expertise to enhance the framework used and to reframe the analytic algorithm used.

[1]secure sockets layer

A
Application Code

```
#include <iostream>
#include <time.h>
#include <stdlib.h>
#include <cstring>

void f1(){
  return;
}

void f2(char* input){
  char buf[8];
  std::memcpy(buf, input, strlen(input)*sizeof(char));
  return;
}

int main(int argc, char* argv[]) {
  struct timespec tStart;
  struct timespec tEnd;

  clock_gettime(CLOCK_MONOTONIC, &tStart);

  if (argc < 3){
    std::cout << "Please enter a number of loop and a dummy input" <<
     std::endl;
    return 0;
  }

  int nb_loop = atoi(argv[1]);
  for (int i = 0; i < nb_loop; i++)
        f1();

  f2(argv[2]);

  clock_gettime(CLOCK_MONOTONIC, &tEnd);

  double elapsed = (tEnd.tv_sec - tStart.tv_sec);
  elapsed += (tEnd.tv_nsec - tStart.tv_nsec) / 1000000000.0;

  std::cout << "Time taken: " << elapsed << std::endl;
    return 0;
}
```

Figure A.1: Application code in C++

B
Test Environment

	Device	SGX Cloud Server	Big Data Framework
Processor	Intel(R) Core(TM) i5-3470 @ 3.20GHz	Intel(R) Core(TM) i5-7500 @ 3.40GHz	Intel(R) Core(TM) i5-7500 @ 3.40GHz
OS	Ubuntu 18.04.1 LTS	Ubuntu 16.04.4 LTS	Debian GNU/Linux 9.5 (Virtual Machine)
RAM			2Go
SWAP			10Go

Table B.1: Hardware Environment

	Device	SGX Cloud Server	Big Data Framework
Boost Library	1.65.1	1.58.0	
Google Protobuf	3.5.1	3.5.1	
SGX SDK		2.3	
Fluentd		1.2.2	
MongoDB			4.0.0
Hadoop			2.7.7
Spark			2.3.1
mongo-spark-connector			2.11:2.3.0

Table B.2: List of library and software version

References

Abera, T., Asokan, N., Davi, L., Ekberg, J.-E., Nyman, T., Paverd, A., Sadeghi, A.-R., & Tsudik, G. (2016). C-flat: Control-flow attestation for embedded systems software.

Arunraj, N. S., Hable, R., Fernandes, M., Leidl, K., & Heig, M. (2017). Comparison of supervised, semi-supervised and unsupervised learning methods in network intrusion detection system (nids) application. *AKWI*.

Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., & Kohno, T. (2011). Comprehensive experimental analyses of automotive attack surfaces. *USENIX Security Symposium*.

Flake, H. (2002). Graph-based binary analysis. Blackhat Conference.

Frank, M. (2018). Internet of babies – when baby monitors fail to be smart. SEC Consult Vulnerability Lab and University of Applied Sciences Technikum Wien.

Gayou, S. (2018). Remote code execution on the smiths medical medfusion 4000. Github.

Hunt, T. (2016). Controlling vehicle features of nissan leafs across the globe via vulnerable apis.

Inoubli, W., Aridhi, S., Mezni, H., Maddouri, M., & Nguifo, E. (2018). A comparative study on streaming frameworks for big data. *44th International Conference on Very Large Data Base*.

Kalan, M. (2015). Tutorial for operationalizing spark with mongodb.

Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., & Hovav Shacham, S. S. (2010). Experimental security analysis of a modern automobile. *IEEE Symposium on Security and Privacy*.

Kovelman, A. (2017). A remote attack on the bosch drivelog connector dongle. Article by Argus Research Team.

Laurenzano, M. A., Tikir, M. M., & amd Allan Snavely, L. C. (2010). Pebil: Efficient static binary instrumentation for linux. *IEEE International Symposium on Performance Analysis of Systems & Software*.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., & Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*.

Miller, C. & Valasek, C. (2010). Remote exploitation of an unaltered passenger vehicle.

Omar, S., Ngadi, A., & Jebur, H. H. (2013). Machine learning techniques for anomaly detection: An overview. *International Journal of Computer Applications*.

Rushanan, M., Rubin, A. D., Kune, D. F., & Swanson, C. M. (2014). Sok: Security and privacy in implantable medical devices and body area networks. *IEEE Security and Privacy*.

S, R. K. & Mary, R. R. (2017). Comparative performance analysis of various nosql databases: Mongodb, cassandra and hbase on yahoo cloud server. *Imperial Journal of Interdisciplinary Research*.

Simões, R. (2009). Apac: An exact algorithm for retrieving cycles and paths in all kinds of graph. *Polytechnical Studies Review*.

Singh, P. & Panda, S. (2018). Threat modeling for automotives. *International Research Journal of Engineering and Technology*.

Thorelli, L.-E. (1966). *An Algorithm for Computing All Paths in a Graph*. Technical report, Scientific Notes - University of Stockholm.

Vaarandi, R. & Niziński, P. (2013). *A Comparative Analysis of Open-Source Log Management Solutions for Security Monitoring and Network Forensics*. Technical report, NATO Cooperative Cyber Defence Centre of Excellence.

Wang, S., Wang, P., & Wu, D. (2016). Uroboros: Instrumenting stripped binaries with static reassembling. *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*.

Xu, L., Sun, F., & Su, Z. (2010). Constructing precise control flow graphs from binaries.

Zonenberg, A. (2016). Remotely disabling a wireless burglar alarm.