On the Relationship between Software Security and Energy Consumption

Miltiadis Siavvas*[†], Charalampos Marantos[‡], Lazaros Papadopoulos[‡],
Dionysios Kehagias[†], Dimitrios Soudris[‡], Dimitrios Tzovaras[†]
* *Imperial College London, London, United Kingdom*
[†] *Centre for Research and Technology Hellas, Thessaloniki, Greece*
[‡] *National Technical University of Athens, Greece*
m.siavvas16@imperial.ac.uk, hmarantos@microlab.ntua.gr, lpapadop@microlab.ntua.gr,
diok@iti.gr, dsoudris@microlab.ntua.gr, dimitrios.tzovaras@iti.gr

***Abstract*—Mitigating software vulnerabilities typically requires source code refactorings for implementing necessary security mechanisms. These mechanisms, although they enhance software security, they usually execute a large number of instructions, adding a performance/energy penalty to the overall application. Conversely, source code transformations are extensively performed by developers in order to improve the runtime quality of applications, in terms of performance and energy efficiency. These transformations may indirectly affect software security, since they may lead to the introduction of new security issues. In this work, we empirically examine the impact of source code-level energy/performance optimizations on software security and vice versa. The preliminary results of the empirical study suggest that the energy-related transformations may indirectly affect software security, whereas the incremental addition of security mechanisms may lead to an important increase in the energy consumption of software applications.**

Keywords—software security, energy consumption, performance, trade offs

I. Introduction

Software security is an aspect of major concern for software applications since the exploitation of a single vulnerability may lead to far-reaching consequences [1]. Most of the software vulnerabilities stem from a small number of common programming errors [2]–[4]. In order to mitigate these errors and protect software applications against critical vulnerabilities, appropriate security mechanisms should be implemented. However, this necessitates source code transformations (i.e., refactorings), which may negatively affect important runtime attributes like performance and energy consumption. This overhead may be important for applications running on platforms that are characterized by restricted energy capacity and computational power, such as embedded systems.

Nowadays there is a large variety of heterogeneous embedded computing architectures that promise to offer increased computing capabilities. However, energy consumption is a very critical parameter for embedded systems. As a result, from the embedded software perspective, a plethora of techniques and tools are used in order to efficiently exploit the characteristics of heterogeneous architectures and resources. At the source code level, transformations are performed in order to improve the memory utilization [5], to increase performance, to eliminate system calls or to accelerate the application utilizing the heterogeneity in a proper way to build accelerators [6]. These transformations may have an important impact on software security, a fact that can not be neglected in order to build secure and reliable applications.

In the present paper, we empirically investigate the inter-relationships between software security and energy consumption. More specifically, we examine whether source-to-source transformations performed for improving software security (i.e., for mitigating existing vulnerabilities) may affect the energy consumption of software applications, as well as whether transformations applied for improving energy consumption may influence software security.

For this purpose, a set of security and energy source code optimizations (i.e., transformations) are applied on a set of software applications retrieved from popular benchmarks. Subsequently, the energy consumption and the security level of the transformed applications are measured using popular tools. The outputs of these tools are used to reach interesting conclusions about the relationships between these two quality attributes. To the best of our knowledge, this is the first study that investigates the interdependencies between software security and energy consumption at the source code level.

The rest of the paper is structured as follows. Section II discusses how the present work advances the current literature, whereas in Section III a set of typical source-to-source transformations for improving energy consumption and software security are presented. Section IV provides a description of the conducted experiments, along with a discussion of their main results. Finally, Section V summarizes the work conducted and concludes the paper.

II. Related Work

The inter-relationships between different quality attributes (i.e., non-functional requirements) is an interesting topic in the software engineering literature. However, design-time and runtime quality attributes are normally studied separately. For instance, a large number of research endeavors can be found in the related literature focusing on the interdependencies between the performance, the energy consumption, and the reliability of software applications [7], which are typical runtime qualities. Similarly, the inter-relationships between design-time qualities, such as Software Security and Maintainability have been individually studied. For example, in a recent study [8], the relationship between Technical Debt [9], which is a measure of software quality, and Software Security was empirically evaluated.

To the best of our knowledge, there are only a few known studies that investigate the relationship between runtime and design-time qualities such as [10], [11] and [12], that emphasize most on Maintainability and Energy Consumption. More specifically, in [12], the authors empirically evaluated the impact that energy-related transformations may have on the Technical Debt of software applications, and conversely the effect that code refactorings for quality improvement may have on energy consumption. Similarly, in the present paper the impact of energy-related transformations on software security, as well as the energy consequences of security-related transformations are empirically examined.

In fact, several studies have investigated the impact of security mechanisms on the energy consumption of software applications. However, existing studies are limited on the energy consumption of individual encryption algorithms [13]–[15], or of broader security protocols [16]–[18], without focusing on the impact that actual security-relevant source code transformations for mitigating common vulnerabilities may have on the energy consumption of software applications. In addition, no research endeavors can be found focusing on the impact that energy-related transformations may have on the security of software applications.

To the best of our knowledge, this is the first study that specifically investigates the inter-relationships between the runtime attribute of energy consumption and the design-time attribute of software security. It should be noted that in the present study we emphasize on Software Security, which is a design-time quality attribute, and not on the runtime security, which is often termed Application Security [2].

III. Transformations for Energy and Security Improvement

The work presented in the present paper is part of the EU-funded H2020 SDK4ED[1] project (Fig. 1). The purpose of the SDK4ED project is to develop a platform that will enable the identification of trade-offs among important design-time and runtime software quality attributes, with emphasis on Maintainability, Security, Reliability, and Energy Consumption. The envisaged platform, through its recommendations, is expected to facilitate developers optimize their code by achieving a satisfactory compromise between these often conflicting quality criteria. In the present section, we present a set of indicative source-to-source transformations for reducing the energy consumption of software application, as well as a set of typical security mechanisms that should be implemented in the source code for enhancing the protection against common vulnerabilities.

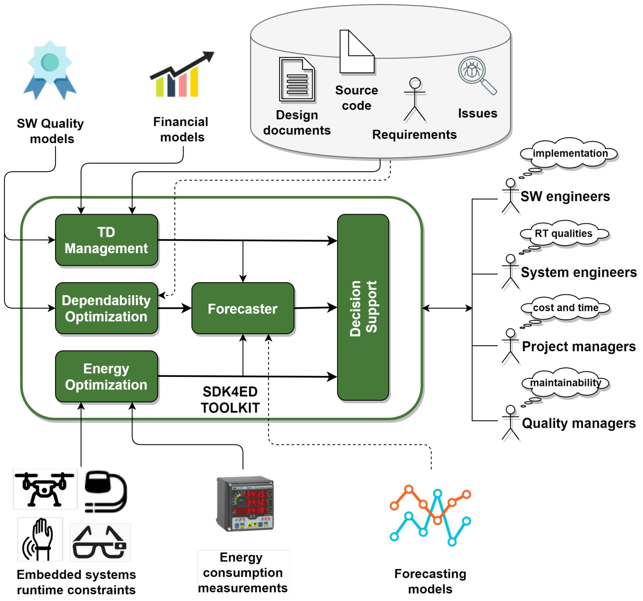

Fig. 1: The high-level overview of the envisaged SDK4ED Toolkit platform.

A. Security Transformations

As already mentioned, software vulnerabilities are normally software bugs with security implications [19]. In Table I we present some indicative examples of common vulnerabilities, along with the source code transformations that are required for their mitigation. It should be noted that emphasis is given on the most popular vulnerabilities, as reported by the OWASP Top 10[2] and CWE/SANS Top 25[3] lists of security risks. The Common Weakness Enumeration (CWE) IDs of each one of the presented vulnerabilities are also provided in order to help the reader find more detailed information. CWE[4] is a dictionary of common weaknesses that can lead to important vulnerabilities.

The first vulnerability presented in Table I is OS Command Injection. This vulnerability arises when the software product constructs an OS command using user-defined (i.e., tainted) inputs, without proper validation or neutralization. This could allow attackers to execute unexpected and dangerous commands directly on the operating system. In the given example, the *p* variable receives user-defined data from a user request, and these data are directly used for executing a command. In order to mitigate this issue, the user-defined input should be checked for illegal characters (i.e., input validation), and in case that illegal characters are present, it should be sanitized by removing these illegal characters (i.e., input sanitization/neutralization). In addition, instead of string concatenation, the final command should be constructed by using a special method (i.e., parameterization). Alternatively, the command and the data could be passed as individual

[1] https://sdk4ed.eu/

[2] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[3] http://cwe.mitre.org/top25/

[4] https://cwe.mitre.org/

parameters to a *ProcessBuilder* object instead of using the *Runtime.exec()* method.

The second vulnerability is the Cross-site Scripting vulnerability. Similarly to OS Command Injection, this vulnerability arises when the software product does not neutralize or incorrectly neutralizes user-supplied inputs before they are placed in output that is used as a web page. This allows attackers to potentially inject malicious scripts into otherwise benign websites, which are then executed at the client side. In the given example, the user-defined variable *data* is written directly to an HTML response, without proper validation and neutralization. This issue is avoided by properly neutralizing the user-defined input by escaping (i.e., encoding) special HTML characters before writing them to the HTML page.

The third vulnerability example is the Buffer Overflow vulnerability, which is popular in unsafe programming languages like C and C++. This security issue arises when the software application does not perform bounds checking and allows input to write beyond the end of an allocated buffer, overwriting in that way adjacent memory locations. These locations may contain data or executable code, leading the program to unexpected behavior including memory access errors, incorrect results and crashes. In the given example, the function *fun()* receives a parameter *str* and copies this parameter to the *buffer* array, without checking their bounds, potentially leading to an overflow. This issue is addressed by properly checking the sizes of the two buffers before performing the copy.

From the above examples it is clear that in order to mitigate vulnerabilities source code transformations are required for adding appropriate security mechanisms. These mechanisms range from simple security-relevant checks to special methods provided by security-relevant APIs (e.g., sanitization algorithms, encoding methods, etc.). In all these cases, additional source code is added to the application, which is expected to have an impact on the performance and the energy consumption of the applications. This is more eminent in the case of library-specific methods, since they usually execute a large number of instructions behind the scenes, adding execution time and energy overheads to the overall application.

Another interesting observation is that in order to achieve sufficient protection against specific vulnerabilities, multiple security mechanisms should be inserted. For instance, in order to achieve sufficient protection against the OS Command Injection vulnerability, the mechanisms of input validation, input sanitization, and parameterization could be applied in conjunction. The incremental addition of security mechanisms is expected to introduce additional overheads to the application.

Although these observations are highly intuitive, they are not supported in the literature by measurements on actual applications. Hence, in the present paper, we attempt to empirically evaluate the impact that the addition of security mechanisms may have on the energy consumption of software applications.

B. Energy Transformations

Performance and energy consumption are very critical parameters for modern embedded devices and consequently, a large variety of methodologies and techniques have been proposed in order to optimize them. Most of them require source code transformations [20]. In this sub-section, we will try to present a set of indicative source-to-source transformations for reducing energy consumption and boosting performance. The discussed transformations are summarized in Table II.

The first category of energy/performance optimization techniques, at application level, aims to improve the memory hierarchy utilization and/or to exploit the parallel processing capabilities of multi-core embedded systems [5]. Since the energy consumed by memory references depends on whether the access hits or misses in the cache memory, we might claim that the cache behaviour is very important for optimizing energy/performance. Typical examples of these techniques are the loop transformations that target to improve the cache performance, through the data locality, as well as to expose parallelism and to reduce the overhead of the loops, that are usually very computationally expensive parts of an application. Furthermore, due to the fact that each memory access has a cost in terms of energy and performance, this kind of transformations aims also to improve the memory utilization and to reduce memory allocation and memory accesses. A simple example is to avoid the unnecessary reassignment of variables. Considering the case that this variable is an array, their impact may be significant. Another typical example of these methods is to switch from dynamic memory allocation to static and vice versa. On the one hand, static memory allocation increases memory requirements. However, in some cases, performance is expected to improve, due to the fact that dynamic memory allocations impose overhead. On the other hand, if the memory requirements are high and taking into account the limited resources of low-cost embedded systems, no proper utilization of the static allocated memory could run the risk of running out of memory.

A massive improvement of the performance and reduction of energy consumption can be achieved by using accelerators [21]. Nowadays, a plethora of heterogeneous embedded computing architectures provides increased performance at constrained energy consumption. A complementary infrastructure, which is part of modern heterogeneous System-on-Chip embedded devices usually includes both CPU and GPU or CPU and FPGA. This kind of chips offers additional computation capabilities and the possibility of performance optimization, through offloading heavy parts of the application to the GPU or FPGA. This is the *second category* of energy transformations presented in this manuscript. Offloading parts of an application on an accelerator cannot be considered as a trivial task, due to the large number of source code refactorings that has to be performed. Depending on the kind of accelerator different tools and programming or hardware description languages have to be used in a proper way. Table II Category 3 presents two very simple examples of adding

TABLE I: Indicative source code transformations for mitigating common vulnerabilities.

Before — **After**

Vulnerability 1: OS Command Injection
Impacts: Confidentiality, Integrity, Availability, Non-repudiation
Relevant CWE(s): CWE-78

```java
...
String p = request.getParameter(”param”);
Runtime rt = Runtime.getRuntime();
rt.exec(”cmd.exe /c echo ” + p);
...
```

```java
...
String p = request.getParameter(”param”);

// Input Validation
if(p.containsIllegalChars()) {

    // Input Sanitization/Neutralization
    String p = SanitizeInput.replaceIllegalChars(p);
}

// Parameterization
String command = ”cmd.exe /c echo %s”;
command.format(”%s”, p);

Runtime rt = Runtime.getRuntime();
rt.exec(command);
...
```

Potential Mitigation: Input Validation; Input Sanitization/Neutralization; Parameterization

Vulnerability 2: Cross-site Scripting (XSS)
Impacts: Confidentiality, Integrity, Availability, and Access Control
Relevant CWE(s): CWE-79, CWE-82, CWE-85, CWE-89, CWE-692

```java
...
String data = request.getParameter(”param”);
if (data != null) {
    response.getWriter().println(”<br>” + data);
}
...
```

```java
...
String data = request.getParameter(”param”);
if (data != null) {
    // Input Escaping/Encoding
    data = StringEscapeUtils.escapeHtml(data);
    response.getWriter().println(”<br>” + data);
}
...
```

Potential Mitigation: Input Escaping/Encoding

Vulnerability 3: Buffer Overflow
Impacts: Confidentiality, Integrity, and Availability
Relevant CWE(s): CWE-120, CWE-129, CWE-131

```c
...
void fun(char* str) {
    char buffer[10];
    strcpy(buffer, str);
}
...
```

```c
...
void fun(char* str) {
    char buffer[10];
    // Bounds Checking
    if (sizeof(str) <= sizeof(buffer)) {
        strcpy(buffer, str);
    }
}
...
```

Potential Mitigation: Bounds Checking

Note: The above examples are written in Java and C/C++ programming languages. However, these snippets are modified for better illustration, and therefore they may not be directly executable. They should be treated as pseudocodes.

vectors kernels in GPUs (CUDA and OpenCL). The whole application needs to be transformed into a version that offers the capability of running in parallel in order to be supported by the kernels. The procedure written in these kernels is executed in all the GPU threads. The required data need to be copied from the host (CPU) to GPU and vice versa. As a result, a large number of transformations in the source code need to be performed.

From the above analysis and the examples presented in Table II, it is obvious that the performance/energy-related transformations do not seem to have a direct impact on software security. More specifically, no vulnerabilities like those presented in Section III-A seem to be directly introduced. However, since code refactoring is required, security issues can be indirectly introduced. In fact, code refactoring is considered an important source of vulnerabilities, as several studies have highlighted that modified code is more prone to security issues [22]. Hence, an indirect impact of energy transformations on software security is expected to be observed.

IV. Experimental Results and Discussion

In the present section, the relationship between software security and energy consumption is empirically evaluated. In particular, we initially investigate the indirect impact that the energy-relevant transformations may have on the security of software applications. Subsequently, we examine the effect that the implementation of common software security mechanisms may have on the energy consumption of the software

TABLE II: Indicative source code transformations for improving energy/performance

Before	After
Category 1: Loop transformations and Memory Optimization **Potential optimizations:** Better cache performance, data locality, exposing parallelism, loop CPU cycles reduction, memory accesses reduction	

1a Loop Merge:
`for (i=0; i<N; i++) {`
`    //do something...`
`}`
`for (i=0; i<N; i++) {`
`    //do something else...`
`}`
	`for (i=0; i<N; i++) {`
	`    //do something...`
	`    //do something else..`
	`}`

1b Loop Tiling:
`for (i=0; i<MAX; i++) {`
`    for (j=0; j<MAX; j++) {`
`        A[i][j] = A[i][j] + B[i][j];`
`    }`
`}`
	`for (i=0; i<MAX; i+=BLOCK_SIZE) {`
	`    for (j=0; j<MAX; j+=BLOCK_SIZE) {`
	`        for (ii=i; ii<i+BLOCK_SIZE; ii++) {`
	`            for (jj=j; jj<j+BLOCK_SIZE; jj++) {`
	`                A[ii][jj] = A[ii][jj] + B[ii][jj];`
	`            }`
	`        }`
	`    }`
	`}`

1c Loop Unrolling:
`for (i=0; i<100; i++) {`
`    A[i] = B[i];`
`}`
	`for (i=0; i<100; i+=4) {`
	`    A[i] = B[i];`
	`    A[i+1] = B[i+1];`
	`    A[i+2] = B[i+2];`
	`    A[i+3] = B[i+3];`
	`}`

1d Unnecessary reassignment removal:
`for (i=0;i<N;i++){`
`    for (j=0;j<N;j++){`
`        a = arr[i];`
`        ...`
`    }`
`}`
	`for (i=0;i<N;i++){`
	`    a=arr[i];`
	`    for (j=0;j<N;j++){`
	`        ...`
	`    }`
	`}`

1e Dynamic to static allocation:
`x=(int*)malloc(10*sizeof(int));`
`if (x==NULL) {`
`    // error message.`
`}`
`free(x);`
	`int x[10];`

Category 2: Offloading on Accelerators of modern heterogeneous devices (e.g. GPUs) **Potential optimizations:** A massive performance/energy improvement through the exploitation of the high parallel processing capabilities of accelerators	

3a Simple OpenCL vector addition kernel example:
`for (i=0;i<N;i++){`
`    c[i] = a[i] + b[i];`
`}`
	`kernel void Add(global const float *a, global`
	`const float *b, global float *c)`
	`{`
	`    int i = get_global_id(0);`
	`    c[i] = a[i] + b[i];`
	`}`

3b Simple CUDA vector addition kernel example:
`for (i=0;i<N;i++){`
`    c[i] = a[i] + b[i];`
`}`
	`__global__ void Add(float *a, float *b, float *c)`
	`{`
	`    int i = blockIdx.x * blockDim.x + threadIdx.x;`
	`    c[i] = a[i] + b[i];`
	`}`

applications. In the following, a detailed discussion of the experiments setup and their main results is provided.

A. Impact of Energy Transformations on Software Security

The purpose of the present experiment is to empirically evaluate the indirect impact that the energy-related source-to-source transformations may have on the security of software applications. More specifically, we examine whether the transformations that are performed on a set of software applications for improving their energy consumption may affect their security level and in which ways.

For the purposes of the present experiment, 17 applications (i.e., benchmarks) were retrieved from the Rodinia Suite [23]. Rodinia is a popular benchmark suite that provides a set of applications for the study of heterogeneous systems. The benchmark provides publicly available programs in multiple versions, coupled with sets of data. Each application has different inherent architectural characteristics, that affect parallelization, synchronization, data transfers, and communication and as a result, the performance and power consumption. Rodinia is a widely-used suite especially by researchers in the field of embedded systems and it is considered as a reliable benchmark for demonstrating, testing and presenting research results and comparisons. Each Rodinia application includes CPU and GPU versions of the same applications. The GPU version exhibits remarkable speed-ups as well as reduced energy consumption compared to the CPU versions. For example, Fig. 2 presents the energy consumption of 5 Rodinia applications with significant savings. The energy was measured in NVIDIA Jetson TX1 Module.

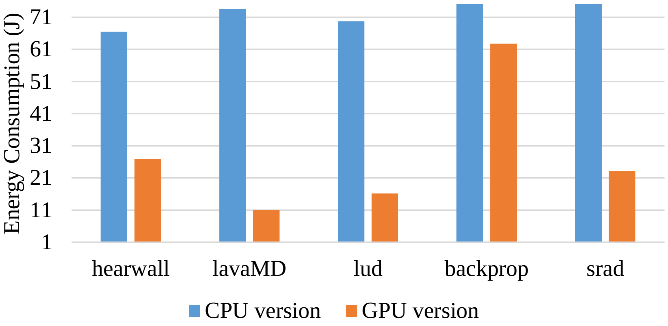

Fig. 2: Energy consumption improvement for the GPU version of some indicative benchmarks of the Rodina suite

In order to assess the security level of the selected applications the CppCheck[5] static analysis tool was utilized. CppCheck is a popular open-source static code analyzer that is able to detect a large number of software bugs, including security-related issues (i.e., potential vulnerabilities) in C/C++ software applications. The tool was properly configured in order to search only for security-relevant issues, and particularly for Buffer Overflows, I/O Issues, String Issues, Null Pointer Dereferences, Resource Handling Issues, and Exception Handling Issues. As an indicator of software security the total number of potential vulnerabilities reported by CppCheck was used. More specifically, the tool was employed to analyze the 17 selected software applications before and after the application of the energy-related source code transformations (i.e., the CPU and GPU versions of the selected Rodinia applications). The total number of potential vulnerabilities detected by the tool are presented in Table III.

As can be seen by Table III, 13 applications have more issues after the code transformations, two have exactly the same number of issues, whereas there are also two applications (namely srad and streamcluster) that the energy-related transformations lead to a reduction in their number of potential vulnerabilities. Hence, we can observe that in the vast majority of the studied applications the energy-related transformations lead to an increase in the number of security issues reported by the CppCheck tool.

TABLE III: The number of security-relevant issues (i.e., potential vulnerabilities) of the applications provided by the Rodinia suite, before and after the performance/energy-relevant transformations (i.e., offloading to GPU), as reported by the CppCheck static code analyzer.

Project Name	**Security Issues (before)**	**Security Issues (after)**
backprop	0	7
bfs	0	6
cfd	0	11
hearwall	3	8
hotspot	0	0
hotspot3D	0	7
kmeans	0	13
lavaMD	4	7
leukocyte	88	90
lud	12	13
myocyte	21	21
srad	18	6
streamcluster	24	20
nn	2	9
nw	0	1
particlefilter	0	14
pathfinder	0	2

In order to reach safer conclusions hypothesis testing is applied. More specifically, the following null hypothesis (along with its alternative hypothesis) is formulated and tested at the 95% confidence interval ($a = 0.05$):

> H_0: *No statistically significant difference is observed between the number of security issues before and after the energy-related transformations.*

> H_1: *A statistically significant difference is observed between the number of security issues before and after the energy-related transformations.*

Since we have a repeated measures test, the paired t-test was used. In order to apply the paired t-test, the differences between the security scores of the applications presented in Table III before and after the acceleration should follow a normal distribution. After applying the Kolmogorov-Smirnov Normality test, we reached the conclusion that the distribution of the differences does not differ significantly from the normal distribution, which means that the paired t-test could be applied. The calculated *p-value* of the paired t-test was found to be 0.02797, which is smaller than the threshold of 0.05. This led us to the rejection of the null hypothesis, indicating that a statistically significant difference exists between the number of security issues that the applications contain before and after the acceleration.

From the above analysis, we can conclude that the energy-related transformations may have an indirect impact on software security, since they may lead to the introduction of new security issues. However, there are also cases in which the number of security issues may be reduced after the acceleration. This situation normally arises when the acceleration requires the removal of software components that happen to be prone to security issues.

[5] http://cppcheck.sourceforge.net/

B. Impact of Security Transformations on Energy Consumption

In the present section, we empirically evaluate the impact that security-relevant source code transformations may have on the energy consumption of software applications. More specifically, we examine whether the additional mechanisms that are incrementally added to a given application for enhancing its protection against specific types of vulnerabilities affect its overall energy consumption. A positive answer to this question will provide empirical support to the intuitive observations that were made in Section III-A, regarding the negative impact of security mechanisms on energy consumption.

In the present experiment, emphasis is given on two specific types of vulnerabilities, namely OS Command Injection and Cross-site Scripting (XSS). Both vulnerabilities are caused by not properly validating and neutralizing user-defined inputs. The reasoning behind the selection of these two types of vulnerabilities is that similar security mechanisms are commonly used for their mitigation. In particular, in the present work we focus on the following commonly applied security mechanisms:

- **Input Validation:** The user-defined data are checked in order to determine whether they contain illegal characters.
- **Input Sanitization:** The user-defined data are properly sanitized (i.e., neutralized) by removing illegal characters.
- **Input Encoding:** The user-defined data (and/or command) are properly encoded before being executed in the command line or added in an HTML page.
- **Parameterization:** String concatenation is avoided.

More information about the aforementioned types of vulnerabilities and their associated security mechanisms can be found in Section III-A. At this point, it should be noted that for the case of Input Encoding, the encoding features provided by the OWASP's ESAPI[6] library were utilized.

For each one of these vulnerabilities two applications were retrieved, one from the OWASP Benchmark[7] and another one from the Juliet Test suite [24], which are popular benchmarks of Java programs with known vulnerabilities. In brief, these applications receive user-defined data as input and they directly use these data (i) for constructing a command that is executed on the command line (OS Command Injection examples), and (ii) for constructing an HTML web page (XSS examples). These four applications were used as the basis of our experiment.

For the purposes of the experiment, the four applications were modified by adding the aforementioned security mechanisms. More specifically, for each one of the four subject applications, the security mechanisms presented in the above-mentioned list were added in an incremental manner. After the addition of each security mechanism, the energy consumption of the application was measured. The measurement was performed (through a power monitoring sensor) on the ARMv8 ARM Cortex-A57 processor. Additional results regarding the CPU cycles and memory accesses were measured using Linux Perf: a hardware performance counter tool used in Linux systems gathering information from the hardware [8]. The results of the analysis are presented in Fig. 3 and Fig. 4.

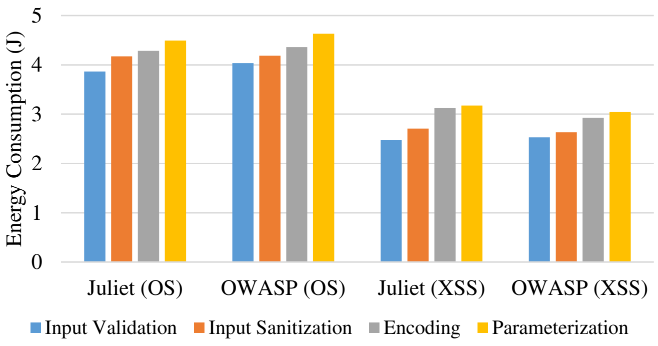

Fig. 3: Impact of Input validation, Parameter Sanitization, Encoding and Parameterization on Energy Consumption

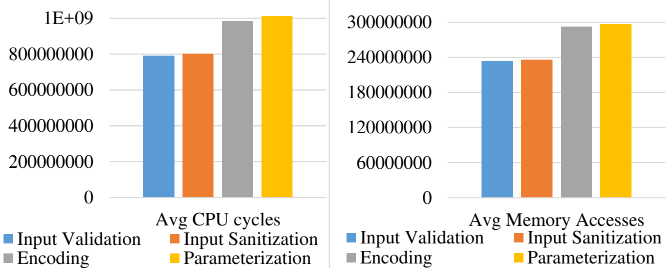

Fig. 4: Impact of Input validation, Parameter Sanitization, Encoding and Parameterization on CPU cycles and memory accesses

As can be seen in Fig. 3, each one of the added security mechanisms leads to an increase in the energy consumption of the software applications. More specifically, we might conclude that a higher energy overhead is added by the Input Encoding check. This is clearly shown in Cross-site Scripting example. Fig. 4 quantifies the average number of execution cycles and memory accesses in order to perform each security check. Taking into account these results, we conclude that the Input Encoding overhead is occurred due to the fact that the memory accesses, and consequently the CPU cycles increase a lot in this step. In order to explain this increment, we have to consider the fact that the encoding features are provided by the ESAPI library. Both the library and the validation properties are loaded from the memory. Based on the analysis depicted in Fig. 3, we might also conclude that for the OS Command Injection checks the Parameterization check has a larger effect in energy consumption compared with the case of XSS. This effect is caused by the utilization of the ProcessBuilder library.

From the above analysis, several observations can be made. First of all, each security transformation (i.e., implemented

[6] https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API#tab=Home

[7] https://www.owasp.org/index.php/Benchmark

[8] https://perf.wiki.kernel.org/index.php/Tutorial

security mechanism) leads to an increase in the energy consumption of the application. Secondly, although the individual mechanisms seem to add small overhead, their incremental addition leads to a significant increase in the overall energy consumption of the application. In fact, the more the security checks that are added (i.e., the stronger the protection of the application against specific vulnerabilities) the higher its energy consumption. Hence, in order to achieve higher security, the energy efficiency of the application should be sacrificed. In platforms with restricted resources (e.g., embedded systems), a trade-off between energy consumption and security could be achieved by reducing the number of implemented security mechanisms, so as to maintain the energy consumption/overhead below a predefined threshold. Finally, the examples presented in this section exhibit a single vulnerability. However, in a large real-world software application the same type of vulnerability may exist in different source code locations. Hence, the implementation of these mechanisms for each one of the existing vulnerabilities are expected to further increase the energy consumption of the broader application.

V. Conclusion

In the present paper we empirically evaluated the inter-relationships between the runtime quality attribute of energy consumption and the design-time attribute of software security. In particular, we initially examined the impact that energy-related source-to-source transformations may have on software security and we discovered a statistically significant difference between the number of security issues that the applications contain before and after the energy optimization (in fact, GPU acceleration). Furthermore, we investigated the impact of security checks on energy consumption, concluding that although each transformation seems to add a small energy overhead, their incremental addition leads to a significant increase in the overall energy consumption.

Several directions for future work can be identified. First of all, a more elaborate empirical study will be conducted in order to investigate the generalizability of the conclusions reached by the present work. In addition, the broader investigation of the inter-relationships between design-time and runtime quality attributes including Software Security, Energy Consumption, and Maintainability is another interesting topic that requires further investigation.

Acknowledgment

This work is partially funded by the European Union's Horizon 2020 Research and Innovation Programme through SDK4ED project under Grant Agreement No. 780572.

References

[1] J. Luszcz, "Apache Struts 2: how technical and development gaps caused the Equifax Breach," *Network Security*, 2018, (under review).

[2] G. McGraw, *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[3] M. Howard, D. LeBlanc, and J. Viega, *24 Deadly Sins of Software Security*. McGraw-Hill, 2010.

[4] M. Siavvas, E. Gelenbe, D. Kehagias, and D. Tzovaras, "Static analysis-based approaches for secure software development," in *Security in Computer and Information Sciences*. Cham: Springer International Publishing, 2018, pp. 142–157.

[5] F. Catthoor, K. Danckaert, K. Kulkarni, E. Brockmeyer, P. G. Kjeldsberg, T. Van Achteren, and T. Omnes, *Data access and storage management for embedded programmable processors*. Springer Science & Business Media, 2013.

[6] C. Baloukas, L. Papadopoulos, D. Soudris, S. Stuijk, O. Jovanovic, F. Schmoll, D. Cordes, R. Pyka, A. Mallik, S. Mamagkakis *et al.*, "Mapping embedded applications on mpsocs: the mnemee approach," in *2010 IEEE Computer Society Annual Symposium on VLSI*, 2010.

[7] S. Cho and R. G. Melhem, "On the interplay of parallelization, program performance, and energy consumption," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 3, pp. 342–353, 2010.

[8] M. Siavvas, D. Tsoukalas, M. Jankovic, D. Kehagias, A. Chatzigeorgiou, D. Tzovaras, N. Anicic, and E. Gelenbe, "An Empirical Evaluation of the Relationship between Technical Debt and Software Security," *9th International Conference on Information Society and Technology*, 2019.

[9] W. Cunningham, "The wycash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[10] R. Verdecchia, R. A. Saez, G. Procaccianti, and P. Lago, "Empirical evaluation of the energy impact of refactoring code smells." in *ICT4S*, 2018, pp. 365–383.

[11] M. F. Oliveira, R. M. Redin, L. Carro, L. da Cunha Lamb, and F. R. Wagner, "Software quality metrics and their impact on embedded software," in *2008 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*. IEEE, 2008, pp. 68–77.

[12] L. Papadopoulos, C. Marantos, G. Digkas, A. Ampatzoglou, A. Chatzigeorgiou, and D. Soudris, "Interrelations between software quality metrics, performance and energy consumption in embedded applications," in *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '18. New York, NY, USA: ACM, 2018, pp. 62–65.

[13] P. Prasithsangaree and P. Krishnamurthy, "Analysis of energy consumption of rc4 and aes algorithms in wireless lans," in *GLOBECOM '03. IEEE Global Telecommunications Conference (IEEE Cat. No.03CH37489)*, vol. 3, Dec 2003, pp. 1445–1449 vol.3.

[14] A. S. Wander, N. Gura, H. Eberle, V. Gupta, and S. C. Shantz, "Energy analysis of public-key cryptography for wireless sensor networks," in *Third IEEE International Conference on Pervasive Computing and Communications*, March 2005, pp. 324–328.

[15] R. Kuchipudi, A. A. M. Qyser, and V. V. S. S. S. Balaram, "An efficient hybrid dynamic key distribution in wireless sensor networks with reduced memory overhead," in *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, 2016.

[16] N. R. Potlapally, S. Ravi, A. Raghunathan, and N. K. Jha, "Analyzing the energy consumption of security protocols," in *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, ser. ISLPED '03. New York, NY, USA: ACM, 2003, pp. 30–35.

[17] N. R. Potlapally, S. Ravi, A. Raghunathan, and N. K. Jha, "A study of the energy consumption characteristics of cryptographic algorithms and security protocols," *IEEE Transactions on Mobile Computing*, vol. 5, no. 2, pp. 128–143, Feb 2006.

[18] D. Fernández-Cerero, A. Jakóbik, D. Grzonka, J. Kołodziej, and A. Fernández-Montes, "Security supportive energy-aware scheduling and energy policies for cloud environments," *Journal of Parallel and Distributed Computing*, vol. 119, pp. 191 – 202, 2018.

[19] N. Munaiah, F. Camilo, W. Wigham, A. Meneely, and M. Nagappan, "Do bugs foreshadow vulnerabilities? An in-depth study of the chromium project," *Empirical Software Engineering*, vol. 22, no. 3, 2017.

[20] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, "The impact of source code transformations on software power and energy consumption," *Journal of Circuits, Systems, and Computers*, vol. 11, 2002.

[21] J. Fowers, G. Brown, J. Wernsing, and G. Stitt, "A performance and energy comparison of convolution on gpus, fpgas, and multicore processors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, p. 25, 2013.

[22] G. J. Holzmann, "The value of doubt," *IEEE Software*, 2017.

[23] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE International Symposium on Workload Characterization*, 2009.

[24] T. Boland and P. E. Black, "Juliet 1.1 C/C++ and java test suite," *Compur (Long. Beach. Calif.)*, vol. 45, no. 10, pp. 88–90, 2012.