Security Evaluation and Hardening of Free and Open Source Software (FOSS)

Robert Charpentier[†]

Mourad Debbabi[‡] and **TFOSS Research Team**[‡][*]

[†]Defence Research and Development Canada, Valcartier, Quebec, Canada
[‡]Computer Security Laboratory, Concordia University, Montreal, Quebec, Canada

Abstract

Recently, Free and Open Source Software (FOSS) emerged as an alternative to Commercial-Off-The-Shelf (COTS) software. Now, FOSS are perceived as a viable long-term solution that deserves careful consideration because of its potential for significant cost savings, improved reliability, and support advantages over proprietary software. However, the secure integration of FOSS in IT infrastructures is very challenging and demanding. Methodologies and technical policies must be adapted to reliably compose large FOSS-based software systems [1]. A DRDC Valcartier-Concordia University feasibility study completed in March 2004 concluded that the most promising approach for securing FOSS is to combine advanced design patterns and Aspect-Oriented Programming (AOP). Following the recommendations of this study a three years project have been conducted as a collaboration between Concordia University, DRDC Valcartier, and Bell Canada. This paper aims at presenting the main contributions of this project. It consists of a practical framework with the underlying solid semantic foundations for the security evaluation and hardening of FOSS.

Keywords: Free and Open Source Software, Security Hardening, Static Analysis, Dynamic Analysis, Aspect Oriented Programming.

1 Introduction

During the past two decades, the software market has been dominated by Commercial-Off-The-Shelf (COTS) products that offer a myriad of functionalities at reasonable prices. However, the intrinsic limitations of COTS software such as security weaknesses, closed source code, expensive upgrades, and lock-in effect have emerged over time. This led to the development of a parallel “economy" based on Free and Open Source Software (FOSS). The latter refers to software whose source code is made available for use and modification without the expensive license fees imposed by COTS software vendors. FOSS is developed either by volunteers, non-profit organizations, or by large computer firms who want to include “commodity" software to give a competitive advantage to their hardware products. To date, thousands of FOSS projects are carried out via Internet collaboration. A plethora of high-quality applications are available for use or modification at no (or small) cost. Many of these FOSS products are widely available and are considered to be as mature as their COTS equivalents. FOSS is now perceived as a viable long-term solution that deserves careful consideration

[*]The TFOSS research team is comprised of: D. Alhadidi, M. Azzam, N. Belblidia, A. Boukhtouta, A. Hanna, R. Hadjidj, H. I. Kaitouni, M. A. Laverdière, H.Z. Ling, S. Tlili, X. Yang, Z. Yang

because of its potential for significant cost savings, improved reliability, and support advantages over proprietary software [2].

Technically, the secure integration of FOSS in IT infrastructures is very challenging and demanding. Methodologies and technical policies must be adapted to reliably compose large FOSS-based software systems [1]. This requirement is exacerbated by the fact that our dependency on software will continue to grow in the next decade. Recent studies confirm that the level of reliability and security currently offered by commercial products is clearly inadequate and that an order of magnitude increase is needed to cope properly with cyber threats [3]. A DRDC Valcartier (Defence R&D Canada Valcartier)-Concordia University feasibility study, completed in March 2004, addressed these issues and considered the technological options to cope with the security and reliability of complex information systems including FOSS and COTS software [2]. It concluded that the most promising approach is to combine advanced security design patterns and Aspect-Oriented Programming (AOP). This facilitates the separation of the definition and implementation of quality and functional specifications [4]. Such a “separation of concerns" will ease the development of secure design patterns to be applied to a wide range of applications. Time and cost investments were also evaluated for the scientific demonstration of these concepts.

Following the recommendations of this study, a three years project has been conducted as a collaboration between Concordia University, DRDC Valcartier, and Bell Canada. This paper aims at presenting the main contributions of this project. More precisely, it presents a practical framework with the underlying solid semantic foundations for the security evaluation and hardening of free and open source software. The evaluation aims to automatically detect vulnerabilities in FOSS that will be corrected by the systematic injection of security code thanks to dedicated aspect oriented technologies. The security code is meant to be derived from security hardening patterns.

The remainder of this paper is organized as follows. Section 2 surveys the related work. In Section 3, we present our first contribution involving static analysis and model checking for detecting security vulnerabilities. Section 4 shows our contribution for security hardening, which is based on aspect orientation. Finally, Section 5 concludes the paper.

2 Related Work

Security code analysis includes security code inspection, automatic analysis and static analysis techniques. Security code inspection techniques are borrowed from software engineering practices [5] and adapted specifically for security purposes. Automatic analysis techniques generally scan the code looking for security sensitive coding patterns that are compiled in checklists. The available techniques are limited to vulnerable coding patterns such as buffer overflows, heap overflows, integer overflows, format string vulnerabilities, SQL injection, cross-site scripting and race conditions [6]. Among the tools that implement these techniques, we can cite: Flawfinder [7], Coverity [8] and PolySpace [9]. Static analysis is used to predict security properties of programs without resorting to their execution. Static analysis techniques include flow-based analysis [10], type-based analysis [11] and abstract interpretation [12]. Finally, the evaluation by security testing is based on the design and execution of test cases in order to identify vulnerabilities in the security features of the software [13, 14].

For FOSS security hardening, four approaches could be distinguished: analyzing, monitoring, auditing, and rewriting [15]. Analysis-based techniques range from simple scanning of code in order to detect malicious code to sophisticated semantics-based analysis of programs. One popular form of analysis-based techniques is certified compilation, which leverages the information generated by the compiler in order to endow the code with a security certificate. This could take the form of proofs as in PCC [16], structured annotations as in ECC [17], or typing annotations with typed assembly languages TAL [18], STAL [19], DTAL [20], Alias Types [21], HBAL [22] and Linearly

Typed Assembly Language [23]. Nevertheless, static analysis is to some extent complex and in some regards undecidable. Monitoring is based on background daemons watching the execution of a program to prevent, at run-time, any harmful operation from taking place [24]. The main drawback of monitoring is the overhead in terms of performance that is induced by the daemons. With auditing-based approaches, the system activity is recorded in an audit trail. This provides a sequence of events related to a trace of program execution and allows to track back any harmful action. If any malicious code causes damage, the audit trail allows to do the recovery and to take the necessary precautions for the future. As of the rewriting-based approach, the code is modified to prevent deviation from the security policies in place. A rewriting tool inserts extra code to perform dynamic checks that ensure that "bad things" cannot happen. Among the research contributions in rewriting-based security, we can cite [25].

In our project, we used aspect orientation as an enabling technology that allows the systematic injection of security in FOSS. Aspect Oriented Programming (AOP) [26] promotes the principle of separation of concerns, thus allowing smooth integration of security hardening mechanisms inside existing software. The most prominent AOP languages are AspectJ [27] and Hyper/J [28], which are built on top of Java programming language. A similar work has also been done to provide AOP frameworks for other languages. For instance, AspectC [29] is an aspect extension of C that is used to provide separation of concerns in operating systems. Similarly, AspectC++ [30] and AspectC# [31] are respectively AOP extensions of C++ and C# languages. Some attempts have been made to use AOP for security. For instance, Cigital Labs conducted a DARPA-funded project [32], where the AOP paradigm was used to address software security. The main outcomes of this project are a security dedicated aspect extension of C called CSAW [32] and a weaving tool. De Win [33] explored the use of AspectJ to integrate security aspects within applications.

3 Static Analysis and Model-Checking for Vulnerability Detection

Our approach brings into a synergy static analysis and model-checking in order to leverage the advantages and overcome the shortcomings of both techniques. The core idea is to utilize static analysis for the automation and the optimization of program abstraction processes. Moreover, programmers take advantage of model-checking techniques to define a wide range of system-specific security properties. As a result, our approach can model-check large software against customized system-specific security properties. Our ultimate goal is to provide a security verification technique for open source software, thus we base our approach on GCC, which is usually a defacto open-source compiler. The language-independent and platform-independent GIMPLE representation [34] of GCC facilitates static analysis by providing easy access to flow, type, and alias information. Being based on GIMPLE, our approach can be extended to support other languages such as C, C++, and Java that GCC accepts as program feeds. For the verification process, we use the Moped model-checker for pushdown systems [35]. The latter are known to efficiently model program execution and inter-procedural behavior. Moped has a procedural input language called Remopla to define programs as pushdown systems. As such, the program abstraction derived from the GIMPLE representation is serialized into Remopla representation. In addition, we enrich program abstractions with Remopla constructs that compute and capture data dependencies between program expressions. Therefore, we are able to detect insidious errors that involve variable aliasing and function parameter passing. Security properties and program Remopla model are input to Moped in order to detect security violations and provide witness paths leading to them.

Moped allows the verification of reachability properties by looking for the reachability of a specific statement in the Remopla code. Though interesting, this capability is not directly sufficient for verifying security properties. In fact, a security property is the description of a pathological behavior in the execution of a program. Such a behavior requires in general an elaborated formalism to be specified and can rarely be stated as the simple reachability of a specific statement in the program. To specify security properties, we use the formalism of *security automata*. A security automaton is a simple automaton with two spacial states: *start* and *error*, and transitions are mapped to instructions or statements in the program to verify. The reachability of the error state in the security automaton when synchronized with the program behaviors is an indication of the occurrence of the pathology. To overcome the limitation of Moped in this regard, we translate a security automaton into a Remopla representation then synchronize it with the Remopla model of the program in question. This comes to synchronizing the pushdown systems of the program and the security automaton. As such, the problem of verifying a security property is translated into detecting the reachability of the error state in the synchronized model.

3.1 Design and Implementation

Fig. 1 depicts the architecture of our security verification environment. The security verification of programs is carried out through different phases including security property specification, static pre-processing, program model extraction, and property model-checking. In the following paragraphs, we describe the input, the output, and the tasks of each of these phases.

- Phase1. Security Property Specification:

  - Input: Security properties.
  - Output: Remopla automata of security properties.

  The first step of our verification process requires the definition of security properties describing what not to do for the purpose of building secure code. We provide users with a tool in order to graphically characterize the security rules that a program should obey. Each property is specified as a finite state automaton where the nodes represent program states and the transitions match program actions. Final states of automata are risky states that should never be reached. To ease the property specification, our tool supports syntactical pattern matching for program expressions and program statements. The graphically defined properties are then serialized into the Remopla language of Moped model-checker.

- Phase2. Static Analysis for Pre-processing:

  - Input: Program GIMPLE representation and security properties.
  - Output: Call-graph and alias information.

  Given a program and a set of security properties to verify, this process conducts call-graph analysis and alias analysis of the program. By considering the required properties, this phase identifies property-relevant behaviors of the analyzed program and discards those that are irrelevant. Besides, we resort to alias analysis in order to limit the number of tracked variables. We only consider variables that are explicitly used in security-relevant operations together with their aliases. All other variables are discarded from the verification process. The static pre-processing phase helps generating concise models that reduce the size of state spaces to explore.

- Phase3. Program Model Extraction:

  - Input: Program source code and specified security properties.
  - Output: Control-flow driven Remopla model or data-driven Remopla model.

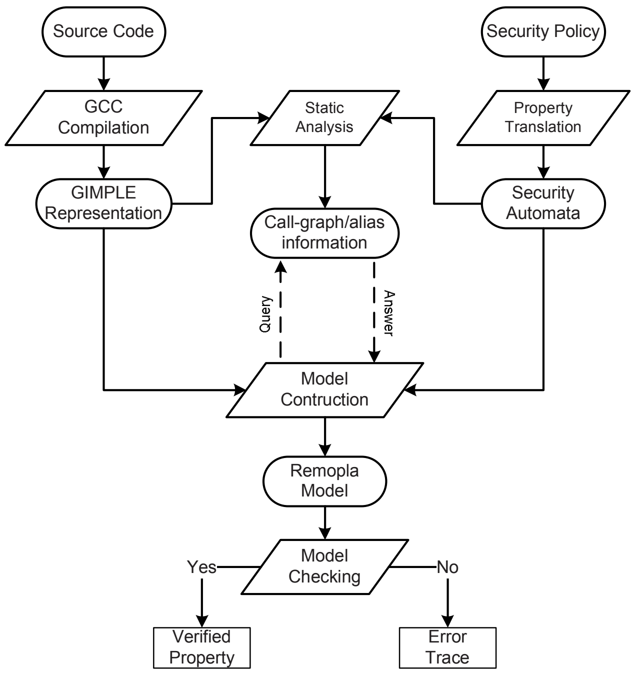


Figure 1: Security Verification Framework

Both the program and the specified properties are translated into Remopla representation and then combined together. The combination of program models and security properties serves the purpose of synchronizing the program behaviors with the security automaton transitions. In other words, transitions in security automata are triggered when they match the current program statement. Our verification approach carries out program model extraction in two different modes: the control-flow driven mode and the data-driven mode. The control-flow mode preserves in the Remopla model the flow structure of the program, but discards data dependencies between program expressions. The resulting Remopla model is efficiently used to detect temporal security property violations and scales to large programs. On the other hand, our data-driven model captures data dependencies between program expressions. Hence, it enhances the precision of our analysis and reduces the number of false positives.

- Phase4. Program Model-Checking:
  - Input: Remopla model.
  - Output: Detected error traces.

Model-checking is the ultimate step of our process. The generated Remopla model is given as input to the Moped model-checker for security verification. An error is reported when a security automaton specified in the model reaches a risky state. The original version of Moped has a shortcoming in a sense that it stops processing at the first encountered error. We have done a modification to Moped in order to be able to detect more than one error in a run. Moreover, we have developed an error trace generation functionality that maps error traces derived from the Remopla model to actual traces from the source code.

3.2 Results and Experiments

This section demonstrates the capability of our security verification framework in detecting real errors in large C software packages. We show that our approach can be efficiently used for uncovering undesirable vulnerabilities in source code. The CERT secure coding website [36] provides a valuable source of information to learn the best practices of C, C++, and Java programming. It defines a standard that encompasses a set of rules and recommendations for building secure code. Rules must be followed to prevent security flaws that may be exploitable, whereas recommendations are guidelines that help improve the system security. The CERT standard also makes another difference between rules and recommendations stating that compliance of a code to rules can be verified whereas the compliance to recommendations is not always verifiable. To assist programmers with the verification of their code, we have integrated in our tool a set of secure coding rules defined in the CERT standard. As such, programmers can use our framework to evaluate the security of their code without the need to have high security expertise. CERT rules can mainly be classified into the following categories:

- *Deprecation rules*: These rules are related to the deprecation of legacy functions that are inherently vulnerable such as `gets` for user input, `tmpnam` for temporary file creation, and `rand` for random value generation. The presence of these functions in the code should be flagged as a vulnerability. For instance, CERT rule `MSC30-C` states the following "*Do not use the `rand()` function for generating pseudorandom numbers*".

- *Temporal rules*: These rules are related to a sequence of program actions that appear in source code. For instance, the rule `MEM3-C` from the CERT entails to *"Free dynamically allocated memory exactly once*". Consecutive free operations on a given memory location represents a security violation. Intuitively, these kind of rules are modeled as finite state automata where state transitions correspond to program actions. The final state of an automaton is the risky state that should never be reached.

- *Type-based rules*: These rules are related to the typing information of program expressions. For instance, the rule `EXP39-C` from the CERT states the following *"Do not access a variable through a pointer of an incompatible type*". A type-based analysis can be used to track violations of these kind of rules.

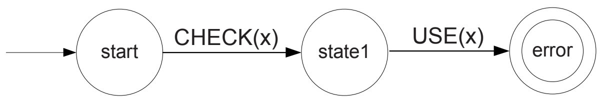
- *Structural rules*: These rules are related to the structure of source code such as variable declarations, function inlining, macro invocation, etc. For instance, rule `DCL32-C` entails to *"Guarantee that mutually visible identifiers are unique*". For instance, the first characters in variable identifiers should be different to prevent confusion and facilitates the code maintenance.

Our approach covers the first two categories of coding rules that we can formally model as finite state automata. In fact, we cover 31 rules out of 97 rules in the CERT standard. We also cover 21 recommendations that can be verified according to CERT. We conducted experiments that consist in detecting the defined set of CERT coding rules against a set of well-known and widely used open-source software. We strive to cover different kinds of security coding errors that skilled programmers can inadvertently produce in their code. The experiments are conducted in the two modes of our security verification tool: the control-flow mode that discards data dependencies and the data-driven mode that establishes data dependencies between program variables.

To illustrate, Fig. 2 gives an example of a security automaton that captures the race condition errors. This security automaton can be used to check the compliance of source code to the following CERT rules:

- `POS35-C`: "Avoid race conditions while checking for the existence of a symbolic link".

- `FIO01-C`: "Be careful using functions that use file names for identification".



CHECK = { access, stat, statfs, statvfs, lstat, readlink, tempnam, tmpnam, tmpnam_r }

USE = { acct, au_to_path, basename, catopen, chdir, chmod, chown, chroot, copylist, creat, db_initialize, dbm_open, dbminit, dirname, dlopen, execl, execle, execlp, execv, execve, execvp, fattach, fdetach, fopen, freopen, ftok, ftw, getattr, krb_recvauth, krb_set_tkt_string, kvm_open, lchown, link, mkdir, mkdirp, mknod, mount, nftw, nis_getservlist, nis_mkdir, nis_ping, nis_rmdir, nlist, open, opendir, pathconf, pathfind, realpath, remove, rename, rmdir, rmdirp, scandir, symlink, system, t_open, truncate, umount, unlink, utime, utimes, utmpname }

Figure 2: Race Condition Automaton (TOCTOU).

The Time-Of-Check-To-Time-Of-Use vulnerabilities (TOCTOU) in file accesses are a classical form of race conditions. In fact, there is a time gap between the file permission check and the actual access to the file that can be maliciously exploited to redirect the access operation to another file. The automaton in Fig. 2 flags a check function followed by a subsequent use function as a TOCTOU error. The analysis results are given in Table 1. The three first columns define the package name, the size of the package, and the program that contains coding errors. The number of reported errors is given in the fourth column (Reported Errors). After inspection of the reported error traces, we classify them into three following columns: column (Err) for potential errors, column (FP) for false positive alerts, and column (DN) for traces that are undecidable with manual inspection. The checking time of programs is given in the last column.

From Table 1, we demonstrate the efficiency and the usability of our approach in detecting real errors in real-software packages. Moreover, our experiment shows that the use of data-driven mode in our framework enhances the analysis precision.

Table 2 summarizes the error traces our tool detected during the experimentation. The properties, the number of reported traces, and the corresponding CERT rules are given in the table, and more details of our experimentation can be found in [37].

4 FOSS Security Hardening

Software security hardening is defined in [38] as *any process, methodology, product or combination that is used to add security functionalities, remove vulnerabilities or prevent their exploitation in existing software*. Security hardening practices are usually manually applied by injecting security code into software [39, 40, 41].

In this section, we address the problems related to the security hardening of FOSS. In this respect, we propose two aspect-oriented and pattern-based approaches for systematic security hardening. The first one is built on top of existing Aspect-Oriented Programming (AOP) technologies while the other one is based on a language-independent and tree-based representation generated by the GNU Compiler Collection (GCC) called GIMPLE. Both approaches are supported by a common structure, which is based on the full separation between the roles and duties of the security experts and the developers performing the hardening. Such proposition constitutes a bridge that allows the security experts to provide the best solutions to particular security problems with the details on why, how and where to apply them. Moreover, it allows the developers to use these solutions to harden open source software without the need to have high security expertise.

Table 1: Results of TOCTOU Analysis.

Package	LOC	Program	Reported Errors	Err	FP	DN	Model-checking time (Sec)
amanda-2.5.1p2	87K	chunker	1	0	1	0	71.6
		chg-scsi	3	2	1	0	119.99
		amflush	1	0	0	1	72.97
		amtrmidx	1	1	0	0	70.21
		taper	3	2	1	0	84.603
		amfetchdump	4	1	0	3	122.95
		driver	1	0	1	0	103.16
		sendsize	3	3	0	0	22.67
		amindexd	1	1	0	0	92.03
at-3.1.10	2.5K	atd	4	3	1	0	1.16
		at	4	3	1	0	1.12
bintuils-2.19.1	986K	ranlib	1	1	0	0	2.89
		strip-new	1	0	1	0	5.49
		readelf	1	1	0	0	0.23
freeradius-server-2.1.3	77K	radwho	1	1	0	0	1.29
inn-2.4.6	89K	nnrpd	1	1	0	0	4.11
		fastrm	1	1	0	0	0.37
		archive	1	0	1	0	0.95
		rnews	1	1	0	0	0.57
openSSH-5.0p1	58K	ssh-agent	2	0	0	2	22.46
		ssh	1	0	1	0	100.6
		sshd	6	3	1	2	486.02
		scp	3	2	0	1	87.95
shadow-4.1.2.2	22.7K	usermod	3	1	0	2	9.79
		useradd	1	1	0	0	11.45
		vipw	2	2	0	0	10.32
		newusers	1	1	0	0	9.2
zebra-0.95a	142K	ripd	1	1	0	0	0.46

Table 2: Summary of Analysis Results.

Experiment Property	Reported Error	Err	FP	DN	CERT Rule
Race Condition	54	33	10	11	POS35-C, FIO01-C
Temporary File Usage	23	23	0	0	FIO43-C
Chroot Jail	2	1	1	0	POS02-C, FIO16-C
Memory Leak	61	11	13	37	MEM-C
Unchecked Return value	14	14	0	0	MEM32-C, EXP34-C
Environment Variable Usage	11	10	1	0	STR31-C, STR32-C, ENV31-C
Deprecated Function	Too many	-	-	-	FIO33-C, POS33-C, MSC30-C

We realize the proposed structure by elaborating a programming independent and aspect-oriented based language for security hardening called *SHL*, developing its corresponding parser, compiler and facilities and integrating all of them into a framework for software security hardening.

In the following, we present the architectures, the design and implementation as well as the results and experiments of each of the aforementioned two approaches.

4.1 Aspect-Oriented Security Hardening

This approach is based on the Security Hardening Language (SHL) that is defined in [42, 43]. We have elaborated an aspect-oriented approach to perform security hardening in a systematic way. In this approach, security experts provide security solutions using an abstract and a general aspect-oriented language called SHL that is expressive, human-readable, multi-language support, and intermediate between English and programming languages. This will relieve developers from the burden of security issues and let them focus on the main functionalities of programs. The security solutions are then applied in a systematic way eliminating the need for manual hardening. The approach provides an abstraction over the actions that are required to improve the security of programs and adopt an aspect-oriented approach to build and develop the solutions.

4.1.1 Architecture

Fig. 3 presents the architecture of this approach. SHL is built on the top of the current AOP technologies that are based on the pointcut-advice model. The solutions elaborated in SHL are expressed by plans and patterns and can be refined into a selected AOP language. Security hardening patterns are high-level and well-defined solutions to known security problems, together with

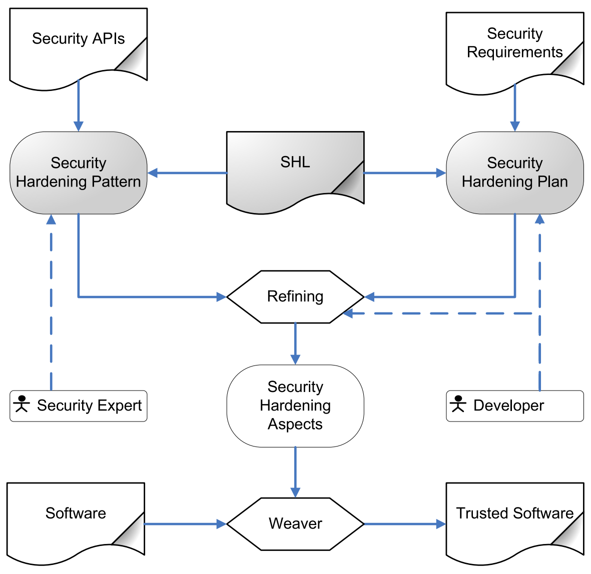


Figure 3: Framework Architecture

detailed information on how and where to inject each component of the solution into an application. Security hardening plans instantiate security hardening patterns with parameters regarding platforms, libraries and languages. The combination of hardening plans and patterns constitutes a bridge that allows security experts to provide the best solutions to particular security problems and allows developers to use these solutions to harden applications by developing security hardening patterns. The development implies refinement of solutions into advices using the existing AOP languages (e.g., AspectJ, AspectC++).

4.1.2 *SHL* Compiler and Framework Implementation

We implemented the `BNF` specification of *SHL* using ANTLR and its associated ANTLRWorks development environment. The generated Java code allows to parse hardening plans and patterns and verify the correctness of their syntax. We built on top of it a compiler that uses the information provided by the parser to build first its data structure, then reacts upon the provided values in order to run the hardening plan and compile and run the specified pattern and its corresponding aspect. This compiler is illustrated in the framework architecture of Fig. 3. Moreover, we integrated this compiler into a development graphical user interface for security hardening. The resulting system provides the user with graphical facilities to develop, compile, debug and run security hardening plans and patterns. It allows also to visualize the software to be hardened and all the compilation and integration activities performed during the hardening. The compilation process is divided into many phases that are performed consequently and automatically. In the sequel, we present and explain these phases.

- *Plan Compilation:* This phase consists of parsing the plan, verifying its syntax correctness and building the data structure required for the other compilation phases. Any error during the execution of this phase stops the whole compilation process and provides the developer with information to correct the bug. This statement also applies on all the other phases.

- *Pattern Compilation and Matching:* A search engine has been developed to find the pattern that matches the pattern instantiations requested in the hardening plan (i.e., pattern name and parameters). A naming convention composed of the pattern name and parameters has been adopted to differentiate between the patterns with same name but different parameters.

Once the pattern-matching the criteria is found, another check on the name and parameters specified inside the pattern is applied in order to ensure that the matching is correct and there is no error in the naming procedure. This includes automatically parsing and compiling the pattern contents to check the correctness of its syntax, verify the matching result and build the data structure required for the running process.

- *Aspect Matching:* Once the pattern is compiled successfully, a search engine similar to the aforementioned one is used to find the aspect corresponding to the matched pattern. However, the additional verification performed in pattern matching is not required here because the aspect will have exactly the same name of the pattern but with different extensions depending on the selected weaver.

- *Plan Running and Weaving:* Plan running is the last phase of the compilation process. Once the corresponding aspect is matched, the execution command is constructed based on the information provided in the data structure, which is built during the previous compilation phases. Afterwards, the aspect is woven with the specified application or module and the resulted hardened software is produced.

- *Aspect Generation:* Aspect generation is an additional feature launched separately to assist the developer during the refinement of a pattern by generating automatically parts of the corresponding aspect. The generated poincuts and advices are enclosed into an aspect that has the same name as the pattern concatenated to its parameters. The developer will have to refine the advices' bodies into programming language code (i.e, C++ or Java) and then run the plan to apply the weaving.

4.2 GIMPLE-based Software Security Hardening

This approach allows applying the security hardening on the GIMPLE representation of software [44]. GIMPLE is an intermediate representation of programs. It is a language-independent and a tree-based representation generated by the the GNU Compiler Collection (GCC) [45] during compilation. GCC is a compiler system supporting various programming languages, e.g., C, C++, Objective-C, Fortran, Java, and Ada. In transforming the source code to GIMPLE, complex expressions are split into three address codes using temporary variables.

Exploiting the intermediate representation of GIMPLE enables to define language-independent weaving semantics that facilitates introducing new security-related AOP extensions. The importance of this stems from the fact that aspect-oriented languages are language dependent. Accordingly, GIMPLE weaving allows defining common weaving semantics and implementation for all programming languages supported by the GCC compiler instead of doing them for each AOP language. For example, instead of having a specific compiler for every aspect-oriented programming language that tries to match join points in code and then does the weaving, the matching and the weaving are done on GIMPLE trees without focusing on a specific programming language. This approach is also based on the aforementioned Security Hardening Language (SHL).

Fig. 4 illustrates the architecture of the GIMPLE weaving approach together with the one presented in Fig. 3. The GIMPLE weaving approach bypasses the refinement step from patterns into AOP languages. The hardening tasks specified in patterns are abstract and support multiple languages, which makes the GIMPLE representation of software a relevant target to apply the hardening. This is done by passing the SHL patterns and the original software to an extended version of the GCC compiler, which at the end generates the executable of the trusted software. For this purpose, an additional pass is added to the GCC compiler in order to interrupt the compilation once the GIMPLE representation of the code is completed. In parallel, the hardening pattern is compiled and a GIMPLE tree is built for each behavior using the routines of the GCC compiler that

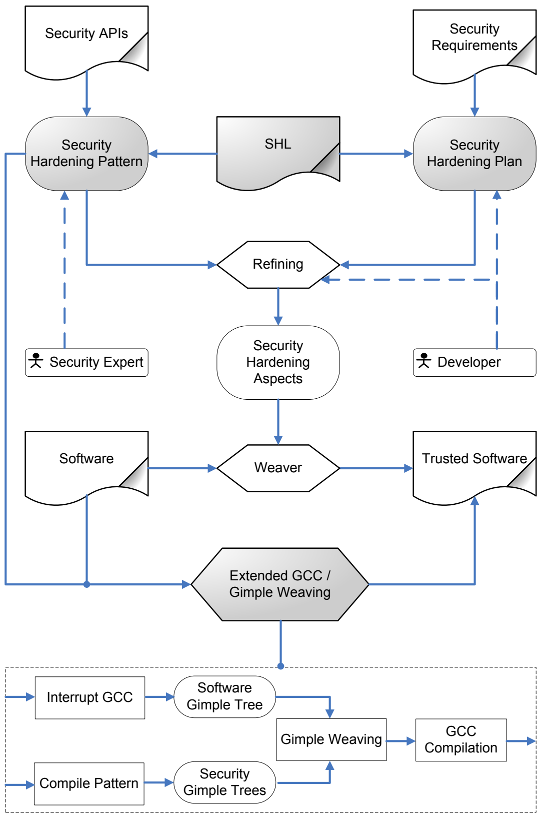


Figure 4: Approach Architecture

are provided for this purpose. Afterwards, the GIMPLE trees generated from the hardening patterns are integrated in the GIMPLE tree of the original code with respect to the location(s) specified in each behavior of the hardening pattern. Finally, the resulted GIMPLE tree is passed again to the GCC compiler in order to continue the regular compilation process and produce the executable of the secure software.

4.2.1 Design and Implementation of Gimple Weaving Capabilities into GCC

We implement into the GCC compiler the weaving features that are inspired from the defined semantics. This implementation allows weaving patterns into the GIMPLE representation of programs before generating the corresponding executables. We handle *before*, *after*, and *replace* behaviors. In addition, we target *call*, *set*, *get*, and *withincode* locations. The implementation methodology that is adopted consists of the following steps. First, we generate a configuration file from the SHL file. This configuration file contains all the information needed for the weaving using our extended GCC. Then, we use the name of this configuration file as an option in a specific command line of the extended GCC compiler. This compiler, which has weaving capabilities, is an extension to the GCC compiler version 4.2.0. Consequently, three input files are needed by the extended compiler to perform the weaving: a source code, a configuration file, and a library containing the subroutines to be woven. In addition to the above option, it is required to specify the library that contains the code to be woven. This is done through GCC's options `-l` and `-L`. Then, a GIMPLE tree is built for the code of each behavior in a pattern. Afterwards, each generated tree is injected in the program tree depending on the insertion point and the location specified in each behavior. Once this weaving procedure is done, the GCC compiler takes over and continues the classical compilation of the modified tree to generate the executable of the hardened program.

4.2.2 Results and Experiments

The main contributions of this approach can be summarized as follows:

- Semantics and algorithms for matching and weaving in GIMPLE are formalized. For this reason, a syntax for a common aspect-oriented language that is abstract and multi-language support and a syntax for GIMPLE constructs are defined.
- Correctness and completeness of GIMPLE weaving are explored from two different views. In the first approach, we address them according to the provided formal matching and weaving rules and the defined algorithms in this paper. On the other hand, we accommodate in the second approach Kniesel's discipline to prove that GIMPLE weaving is correct and complete just in some specific cases because of behavior interactions and interferences.
- Implementation strategies of the proposed semantics are introduced. To explore the viability and the relevance of the defined approach, case studies are developed to solve the problems of unsafe creating of chroot jail, unsafe creating of temporary files, and using deprecated functions.

5 Conclusion

In this paper, we presented an innovative framework for security evaluation and hardening of free and open-source software. For security evaluation, first a vulnerability detection approach has been proposed. This approach brings into a synergy the static analysis and the model-checking in order to leverage the advantages and overcome the shortcomings of both techniques. We demonstrated the efficiency and the usability of our approach in detecting real errors in real-software packages. Moreover, our experiment shows that the use of data-driven mode in our framework enhances the analysis precision. It is important to mention that we have also developed a second approach to detect security vulnerabilities that is based on security testing and code instrumentation. This approach has not been detailed in this paper for the lack of space. Finally, we have presented a security hardening approach. This approach is based on the Security Hardening Language (SHL) that we have defined in [42, 43]. The approach aspect oriented and performs security hardening in a systematic way. In this approach, security experts provide security solutions using an abstract and a general aspect-oriented language called SHL that is expressive, human-readable, multi-language support, and intermediate between English and programming languages. The use of this language relieve developers from the burden of security issues and let them focus on the main functionality of programs. The approach provides an abstraction over the actions that are required to improve the security of programs and adopt an aspect-oriented approach to build and develop the solutions.

References

[1] T. Bollinger. Use of Free and Open-Source Software (FOSS) in the U.S. Department of Defense. Technical Report MP 02W0000101 v1.2.04, MITRE, January 2003.

[2] R. Charpentier and R. Carbone. Free and Open Source Software: Overview and Preliminary Guidelines for the Government of Canada, March 2004. Defence Research and Development Canada – Valcartier.

[3] A. Fecteau and J. P. Rodrique. Certifying Critical Software: JACC Market Survey. Technical report, Geo Alliance International Inc, June 2003.

[4] Mourad Debbabi, Zahia Aidoud, and Ali Faour. On the inference od structured recursive effects with subtyping. *Journal of Functional and Logic Programming*, 1997(5), 1997.

[5] M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3), 1976.

[6] L. Grenier. Practical Code Auditing. *OpenBSD Journal*, December 2002.

[7] D. A. Wheeler. FlawFinder. `http://www.dwheeler.com/flawfinder/`, 2001.

[8] Coverity. Coverity Prevent for C and C++. `http://www.coverity.com/main.html`.

[9] PolySpace. Automatic Detection of Run-Time Errors at Compile Time. `http://www.polyspace.com/`.

[10] C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson. Static Analysis for Secrecy and Non-Interference in Networks of Processes. *Lecture Notes in Computer Science*, 2127:27–41, 2001.

[11] L. Cardelli, A. Gordon, and G. Ghelli. Secrecy and Group Creation. In Ted Hurley, Mícheál Mac an Airchinnigh, Michel Schellekens, and Anthony Seda, editors, *Electronic Notes in Theoretical Computer Science*, volume 40. Elsevier, 2002.

[12] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[13] P. Herzog. *Open-Source Security Testing Methodology Manual.* Institute for Security and Open Methodologies (ISECOM), August 2003.

[14] J. Wack, M. Tracy, and M. Souppaya. Guideline on Network Security Testing. NIST Special Publication 800-42, National Institute of Standards and Technology (NIST), October 2003.

[15] G. McGraw and G. Morrisett. Attacking Malicious Code: A Report to the Infosec Research Council. *IEEE Software*, 5(17), September/October 2000.

[16] G. Necula. Proof-Carrying Code. In *24th POPL*, pages 106–119, Paris, France, January 1997.

[17] D. Kozen. Efficient Code Certification. Technical Report 98-1661, Computer Science Department, Cornell University, January 1998.

[18] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[19] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-Based Typed Assembly Language. In *tic*, volume 1473 of *Lecture Notes in Computer Science*, pages 28–52, Kyoto, Japan, March 1998. springer.

[20] H. Xi and R. Harper. Dependently Typed Assembly Language. Technical Report OGI-CSE-99-008, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, July 1999.

[21] F. Smith, D. Walker, and G. Morrisett. Alias Types. *Lecture Notes in Computer Science*, 1782:366+, 2000.

[22] D. Aspinall and A. B. Compagnoni. Heap Bounded Assembly Language. *Journal of Automated Reasoning*, 31:261–302, 2003.

[23] J. Cheney and G. Morrisett. A Linearly Typed Assembly Language. Technical Report 2003-1900, Department of Computer Science, Cornell University, 2003.

[24] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability Classes for Enforcement Mechanisms. Technical report TR2003-1908, Cornell University, Computing and Information Science, Ithaca, New York, August 2003.

[25] A. Rudys and D. S. Wallach. Enforcing Java Run-Time Properties Using Bytecode Rewriting. In *Proceedings of the International Symposium on Software Security*, Tokyo, Japan, November 2002.

[26] G. Kiczales, J. Lamping, A. Menhdhekar, Ch. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[27] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In *Proceedings of the 2001 European Conference on Object-Oriented Programming (ECOOP'01)*, 2001.

[28] P. Tarr and H. Ossher. HyperJ User and Installation Manual, 2000.

[29] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Proceedings of Foundations of software Engineering*, Vienne, Austria, September 2001.

[30] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems*, Sydney, Australia, February 2002.

[31] H. Kim. AspectC#: An AOSD implementation for C#. Technical Report TCD -CS2002-55, Department of Computer Science, Trinity College, Dublin, 2002.

[32] Cigital Labs. An Aspect-Oriented Security Assurance Solution. Technical Report AFRL-IF-RS-TR-2003-254, Cigital Labs, Dulles, Virginia, USA, Oct 2003.

[33] B. De Win, F. Piessens, W. Joosen, and T. Verhanneman. On the Importance of the Separation-of-Concerns Principle in Secure Software Engineering. Workshop on the Application of Engineering Principles to System Security Design, Boston, MA, USA, November 6–8, 2002, Applied Computer Security Associates (ACSA), 2002.

[34] Diego Novillo. Tree SSA: A New Optimization Infrastructure for GCC. In *Proceedings the GCC Developers Summits3*, pages 181–193, May 25-27 2003.

[35] Stefan Kiefer, Stefan Schwoon, and Dejvuth Suwimonteerabuth. Moped - a model-checker for pushdown systems. (Date of Access: January 20, 2009).

[36] CERT Secure Coding Standard. http://www.securecoding.cert.org, April 2009.

[37] Syrine Tlili, XiaoChun Yang, and Mourad Debbabi. Verification of CERT secure coding rules: Case studies. In *International Symposium on Information Security*. Springer Verlag, 2009.

[38] Azzam Mourad, Marc-André Laverdière, and Mourad Debbabi. Security hardening of open source software. In *Proceedings of the 2006 International Conference on Privacy, Security and Trust (PST 2006)*. McGraw-Hill/ACM, 2006.

[39] Matt Bishop. How attackers break programs, and how to write more secure programs, 2005. Available at `http://nob.cs.ucdavis.edu/~bishop/secprog/sans2002/index.html` (accessed on 2008/11/11).

[40] Michael Howard and David E. LeBlanc. *Writing Secure Code.* Microsoft, Redmond, WA, USA, 2002.

[41] Robert C. Seacord. *Secure Coding in C and C++.* SEI Series. Addison-Wesley, 2005.

[42] Azzam Mourad, Marc-André Laverdière, and Mourad Debbabi. A high-level aspect-oriented based language for software security hardening. In *Proceedings of the International Conference on Security and Cryptography (Secrypt)*, Barcelona, Spain, 2007.

[43] Azzam Mourad, Marc-André Laverdiere, and Mourad Debbabi. Towards an aspect oriented approach for the security hardening of code. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops, AINAW '07*, pages 595–600. IEEE, 2007.

[44] GIMPLE-GNU Compiler Collection (GCC) Internals. Available at `http://developer.apple.com/DOCUMENTATION/DeveloperTools/gcc-4.0.1/gccint/GIMPLE.html` (accessed on 2009/6/1).

[45] GCC-the GNU Compiler Collection. Available at `http://gcc.gnu.org/` (accessed on 2009/6/1).