SIDAN : a tool dedicated to Software Instrumentation for Detecting Attacks on Non-control-data

Jonathan-Christofer Demay, Eric Totel and Frédéric Tronel
SSIR Laboratory, SUPELEC
Rennes, France
first_name.last_name@supelec.fr

Abstract

Anomaly based intrusion detection systems rely on the build of a normal behaviour model. When a deviation from this normal behavior is detected an alert is raised. This anomaly approach, unlike the misuse approach, is able to detect unknown attacks. A basic technique to build such a model for an application is to use the system call sequences of a process. To improve the accuracy and completeness of this detection model, we can add information related to the system call, such as its arguments or its execution context. But even then, attacks that target non-control-data may be missed and attacks on control-data may be adapted to bypass the detection mechanism using evasion techniques.

We propose in this article an approach that focuses on the detection of non-control-data attacks. Our approach aims at exploiting the internal state of a program to detect a memory corruption on non-control-data that could lead to an illegal system call. To achieve this, we propose to build a data-oriented detection model by statically analyzing a program source code. This model is then used to instrument the program by adding reasonableness checks that verify the consistent state of the data items the system calls depends on. We thus argue that it is possible to detect a program misuse issued by an non-control-data attack inside the program during its execution. While keeping a low overhead, this approach can thus lead to the detection of a whole range of non-control-data attacks.

1. Introduction

Any attack on a particular program aims at some point to corrupt some data within its memory space. There are two kinds of data items and therefore an attack can be classified in two types: control-data attack or non-control-data attack. A control-data attack aims to corrupt the memory to modify the execution flow of the program. The return address on the stack or the function pointers in the exception handler are items that may be the target of such an attack. For example, by corrupting these, it is possible to perform an injection code attack (shellcode) or an out-of-context code attack (return to library), both resulting in the execution of code located on an invalid and therefore incorrect path. A non-control-data attack aims to corrupt the memory used by the program to perform its computations. Data items used to compute conditional evaluations or parameters of system calls may be the target of such an attack. By corrupting these, it is possible to make an incorrect use of a valid code. For example, it is possible to force the execution flow to deviate to a valid but incorrect path or to use a correct and valid path to perform system calls with incorrect parameters.

To detect attacks, anomaly-based intrusion detection systems rely on a model that reflects the normal behavior of an application. An alert is generated when a deviation from this normal behavior is detected. Unlike the misuse approaches that rely on the knowledge of the attacks, the anomaly approaches provide a way to detect unknown attacks. The model of this normal behavior can either be built statically (from the specifications [1] or the source code [2]) or dynamically [3]. Numerous Host Intrusion Detection Systems that aim to protect programs against attacks use an anomaly detection approach. Among them, numerous IDS build their model using the approach introduced by Forrest et al. [4] which consists in analyzing the system call sequences of processes. Several enhancements of this approach have been proposed, notably by adding information that is available at the interface level between the system and the process, such as the system call arguments [5] or the execution context [6]. These enhancements improve both detection accuracy and completeness, allowing this approach to detect various attacks. However, numerous non-control-data attacks are still not detected by these enhanced approaches [7] and attacks on control-data and non-control-data may still be adapted to bypass the detection system using evasion techniques [8], [9].

In this paper, we propose an approach for intrusion detection that focus on non-control-data attacks by building a model to detect memory inconsistencies. Since non-control-data attacks need to force specific values for specific data items, they may put the internal state of a program in an inconsistent state regarding the program’s specifications. The work we propose consists in exploiting the internal state of a program to decide if the system calls are valid or not. By checking at various points in a program the consistency of
specific data items, we may detect deviation that could lead to an illegal system call. We thus argue that it is possible to detect a program misuse inside the program during its execution.

To present our approach, we first cover in this paper the related work. Then we present our model for intrusion detection and how we address the problems faced by our approach. Finally we evaluate the effectiveness of our approach against non-control-data attacks and the performance overhead the detection process induces, followed by the concluding remarks and the future work that we are currently considering.

2. Related Work

The anomaly approach for intrusion detection which consists in analyzing the system call sequences of a process [4], even enhanced by taking the arguments [5] and the context [6] of the call into account, is a general approach with a high granularity level. Therefore, it targets a wide range of control-data and non-control-data attacks but is more prone to false positives (false alerts) and false negatives (intrusions missed). Nowadays, control-data attacks and non-control-data attacks are exploiting software vulnerabilities in a different way. Therefore, several approaches have been proposed to specifically improve the detection accuracy and completeness while focusing on either one of them.

Since control-data attacks are currently dominant in real world exploitations of vulnerabilities, many approaches have been proposed to improve the detection of this particular kind of attack. For example Instruction Set Randomization [10] is a technique that randomizes the instructions understood by the processor. Any injected code will fail to execute as planned since the attacker cannot guess the seed used to generate this particular set of instructions. Moreover, Control-Flow Integrity [11] and Program Shepherding [12] are generic techniques that ensure the integrity of the execution flow of a program. They compute a control-flow graph of the program and then use it at run time to enforce the integrity of the program’s execution flow. Because mimicry attacks still need to force the program execution flow to deviate from any valid execution path, they are detected by these approaches. In fact, any deviation from the control-flow graph is prevented, therefore any practical attack on control-data are detected by these approaches. However, all those techniques are completely inefficient against non-control-data attacks since those attacks are performed within an existing valid execution path.

As suggested by Chen et al [7], control-data attacks may be dominant not because of the difficulty to perform control-data attacks on real world vulnerabilities, but because execution flow integrity techniques are rarely deployed and therefore attackers lack the incentive to do otherwise. In fact, they have shown that non-control-data attacks are realistic threats. Therefore, in a situation where execution flow integrity techniques were to be deployed, attacks on non-control-data may be used to bypass these detection mechanisms. Thus, some other approaches have been proposed to address the detection problem of non-control-data attacks.

Data Space Randomization [13] propose to address this problem by randomizing the representation of the data stored in memory. This can be implemented by encrypting and decrypting each data item with a unique random encryption key each time this data item is respectively read or written. When an attacker corrupts a data item within the memory space of a process, since he cannot guess the encryption key used to store the targeted data in memory, he will not be able to force a particular value that may be needed to perform an attack. While this approach may target a wide range of non-control-data attacks, it does not work against attacks that can be performed by forcing a specific data item to diverge from a particular value, like the do_authentication attack used by Chen et al. [7] on OpenSSH. The approach we propose may be able to detect this kind of non-control-data attack that Data Space Randomization will miss.

Other approaches have been proposed. For example Data-Flow Integrity [14] and Write-Integrity Testing [15] are generic techniques that ensure the integrity of the data flow of a program. They compute a data-flow graph that contains for each data read by an instruction the set of instructions that may have written its current value. This data-flow graph is then used at run time to check the integrity of the data flow of a program. If a vulnerability within a program is exploited to corrupt some data, the next time this data is read a deviation from the data-flow graph will be detected thus preventing the intrusion from being effective. These approaches are very effective against all kind of non-control-data attacks, but their major drawback is the overhead they imply at runtime. With the approach we propose, most of the computations are performed during the actual instrumentation and the runtime overhead is therefore very low.

3. Intrusion Detection

3.1. Attacks on Non-control-data

As opposed to attacks on control-data whose goal is to execute unvalid code, attacks on non-control-data aim at exploiting existing valid code in an illegal way by corrupting data items used by this code. As shown by Chen et al. [7], the severity of the compromises permitted by such attacks can be equivalent to that of control-data attacks on various real world vulnerabilities.

For example, an integer overflow vulnerability [16] was discovered in 2001 in various SSH server implementations, including the widely used open-source OpenSSH. In this case, the integer which is subject to an overflow vulnerability
is used to compute a mask. This mask is later used on an offset to prevent a write operation on a buffer from being out of boundary. An attacker can use this integer overflow to bypass the boundary and therefore use it to write arbitrary data at any memory location in the process. Pekka and Kalle [17] have provided a detailed analysis of this vulnerability.

As shown by the code in Figure 1, we could use this vulnerability (which is located in packet_read) to overwrite the value of pw->pw_passwd during do_authloop with an empty string so that we can successfully log into the system with any known account (root would be a good choice) without having to provide any valid password. So in case control-data attacks were to be prevented by a detection mechanism, a malicious user could fall back to a non-control-data attack to successfully exploit this vulnerability. A more detailed explanation on how to exploit this vulnerability in such a way has been explained by Starzetz [18].

However, such attack puts the internal state of the program in an inconsistent state. If the password was empty in the first place, then the execution should not have reached do_authloop and would have been directed to do_authenticated. So when auth_password is called within do_authloop, pw->pw_passwd should be anything but an empty string. Had an executable assertion checked the state consistency of pw and had been inserted right before auth_password, then such an attack would have been detected (see Figure 2, line 11). In fact, we believe that if various checks like this one were to be inserted at different points in the program, a whole range of non-control-data attacks could be detected.

3.2. Data oriented detection Model

The attack we have illustrated on previous section shows that it is possible to detect attacks on data inside the software by inserting checks on data values. The objective of this section is to show that it is possible to build automatically a data oriented behavior model to detect such attacks.

The most easy way to build a program normal behaviour model is to have a precise description of the program specification, and to check if the program deviates from its known specification. However, such an approach requires that the programmer builds a useable model of the specification, which is often not done. However, as soon as we assume the program is correct, the specification could be built from the program source. Such an approach would produce a behavior model that could detect a program misuse, but could miss intrusions that are the result of program faults. In our work, we suppose that the attacks use a correct program to realize illegal actions. We can thus use a program source code to build the behavior model.

We can consider that a program internal state is a set of variables. These variables can be linked together (e.g., a variable must be equal with another one), or have particular properties (e.g., a variable has a defined set of values). These properties are called invariants, and are the base of our model. Our goal is to enhance the detection capability of conventional control flow detection methods (based on system call checking) by focussing on data correctness.

Our approach consists in finding properties in a program to detect data corruption induced by misuses. However, unlike other methods, we do not intend to check properties defined for all the variables in the program. In practice, we try to define a subset of the variables of the program that can have an influence on system calls. This subset defines the set of variables that can influence a system call. Thus, from an intruder point of view, these variables are the target of its attacks, all other variables having no impact on the call.

A system call depends on two types of data: the arguments that are passed as parameters and the value of variables in...
the program that have contributed to the execution of this call (e.g., a variable used in a condition that has led to the execution of this system call). We can thus define for a given system call $SC_i$ its data behaviour model by a triple $(SC_i, V_i, C_i)$ where $V_i$ is the set of variables the system call depends on, and $C_i$ the set of constraints on these variables that can be deduced from the program analysis. We can then define the program normal data behavior model by the set of all $SC_i$, $DBM = \{vi, SC_i\}$.

The object of an attack is to modify the value of one or several variables, and will thus probably break some constraints. An execution of a program is then considered correct if all constraints remain true, and an intrusion is detected if at least one of the constraints of one $SC_i$ is not verified. There are now two points to answer to build this model: how to define the set of variables on which a system call depends on, and how to obtain the invariants that are the constraints that must be verified on these variables at runtime.

### 3.2.1. Building the variables set.

In order to build $V_i$, we must be capable of discovering all variables the system call $SC_i$ depends on. There are several dependencies that can be identified for a system call: data dependency and control dependency. The data dependency relation aims at identifying the variables that influence the value of the arguments of the system call. The control dependency relation aims at identifying the variables that have an influence on the program control flow and that have contributed to the execution of the system call.

In the static analysis field, discovering these relations can be done by using program slicing techniques [19]. A program slice can be defined as the parts of a program that potentially affect the values computed at some point of interest of a program. In our case, we are looking for all variables that influence a system call, and thus all variables used in the system call slice. Generally, slices are computed by computing consecutive sets of transitively relevant statements, according to data flow and control flow dependencies. An alternative method consists in computing a program dependency graph (PDG) [20]. The PDG is a directed graph with vertices corresponding to statements and control predicates, and edges corresponding to data and control dependencies. The slicing problem can be rephrased as a reachability problem in this graph.

In the implementation we propose, we use the dependency graph notion to determine the variables that potentially influence a system call.

### 3.2.2. Constraint discovery.

Once the variables a function call depends on have been determined, we are interested in deriving automatically constraints on these variables. Remember that these constraints together with the set of variables they hold on will help us building executable assertions. To build the set of constraints $SC_i$ we have chosen to use automatable technics coming from the field of static analysis. More precisely, we rely on abstract interpretation [21] to automatically derive from the source code the definition domains of a large class of variables used in a program so as to build constraints of the type e.g "integer variable $x$ lies within the domain $[0,5]$ in all executions". However, it is well known that computing whether a program satisfies a given property or not is an undecidable problem (Rice’s theorem). Of course this very general impossibility result fully applies to the problem of computing the definition set of variables we are trying to solve.

In order to circumvent this problem, various techniques have been proposed among which abstract interpretation is one of the most powerful. As its name indicates, it consists in abstracting the variation domain of objects that are to be studied to make the computation of properties on the selected objects more tractable. The whole art of abstract interpretation is to find the correct level of abstraction that on one side maintains the tractability of the problem, while providing sufficiently precise properties to be useful on the other side. When on such abstraction has been defined, the problem of computing the semantics of a program is translated back into computing the possible traces of the same program but this time in terms of abstract properties rather than fully concrete properties. Once this step has been completed, abstract properties are translated back into concrete properties. Note that hence we need to be able to translate from concrete properties to abstract ones and conversely.

A trivial example of abstraction for integer variables could be the "even" or "odd" property. It is always possible to decide if a variable is odd or even when the permitted set of operations on integers is restricted to the addition, subtraction and multiplication. This property is a toy example since it is rarely useful in real applications. It only serves as an illustration.

An important step in the process of abstraction is to maintain the soundness of the computed semantics. Informally this means that the process of abstraction should preserve the property of the concrete semantics. A theoretical framework has been proposed and formalized by [21] to preserve soundness. This framework is based on monotonic functions on partially ordered sets.

More precisely, let us denote $C$ the set of concrete values, and $P(C)$ the power set of $C$ partially ordered by inclusion ($\subseteq$). Let $\mathcal{A}$ be the set of abstract properties partially ordered by the relation $\subseteq$. Let $\gamma$ be the function that maps each abstraction (an element of $\mathcal{A}$) to the subset of concrete values it represents, and conversely, let $\alpha$ be the function that maps each subset of concrete values to the abstract value that best describes it. These two maps are central in abstract interpretation. Indeed, $\alpha$ describes the abstraction
process. Given a set of concrete values it maps them on a set on abstract values. Conversely, $\gamma$ describes the process of concretization by mapping an abstract value back to the set of concrete it corresponds.

Obviously doing static analysis will require to go back and forth between concrete and abstract values. This suggests that the composition of both functions ($\gamma \circ \alpha$ and $\alpha \circ \gamma$) should satisfy some properties. When $\alpha$ and $\gamma$ are monotonic functions on $P(C)$ and $A$ respectively, [1] have shown that the following properties must hold:

$$\begin{align*}
\forall S \in P(C) & \quad S \subseteq \gamma \circ \alpha(S) \\
\forall a \in A & \quad \alpha \circ \gamma(a) \subseteq a
\end{align*}$$

The first property guarantees that process of abstraction is safe in the sense that it is an over approximation. More precisely the abstraction of a set of concrete values returns an abstract value that best describes a set of concrete values. Conversely, the second property states the process of concretization is an under approximation. This second property is usually restricted to a strict equality, that is $\alpha \circ \gamma(a) = a$, meaning that all abstract properties are exact. If we get back to the parity example, we can define $A$ as being the set $\{\bot, \text{even, odd, } \top\}$, equipped with the following partial order relation:

![Partial Order Relation](image)

Note that $A$ is a lattice. $\gamma$ is then defined as follows :

$$\begin{align*}
\gamma(\text{even}) &= \{-\infty, \ldots, -4, -2, 0, 2, 4, \ldots, +\infty\} \\
\gamma(\text{odd}) &= \{-\infty, \ldots, -3, -1, 1, 3, \ldots, +\infty\} \\
\gamma(\top) &= [-\infty, +\infty] \\
\gamma(\bot) &= \emptyset
\end{align*}$$

Conversely $\alpha$ is defined as follows :

$$\begin{align*}
\beta(2n) &= \text{even} \\
\beta(2n+1) &= \text{odd} \\
\alpha(S) &= \bigcup \{\beta(v) \mid v \in S\}
\end{align*}$$

Using this abstraction on simple programs will produce an over approximation of its concrete semantics. The nice property indues by this formalization is that being an over approximation, it never forgets properties exhibited by the concrete semantics of a program. Hence the properties computed by abstract interpretation can directly be used to prove fact about programs. More elaborated examples of abstract interpretation are typing inference, interval analysis (for integer variables), etc. For this paper, we have mainly use refinements of interval analysis of integer variable. This technics allows to compute interval of values of the form $[i, j]$ with $i \leq j$ for each integer variable.

Abstract interpretation is well suited for this kind of analysis and perform well on code written for embedded device. Indeed, the coding style used in this context is usually characterized by the absence of dynamic memory allocation and the frequent use of integer variables. For the moment being, abstract interpretation does not perform very well on codes that make a high use of pointers, dynamic memory allocation and complex data structures.

### 3.3. Defensive Programming

In order to ensure that data items from the internal state of the program are in a consistent state, we propose to use techniques issued from the dependability domain [11], [22], [23], and more precisely defensive programming. In order to detect errors during the execution, this technique requires to insert state checks in the program that are called reasonableness checks. These checks may be redundant from a specification point of view, but they may also detect when a program has diverged from those specifications, for example when a vulnerability has been successfully exploited. These checks can concern either the program execution flow (to verify that the correct instructions are executed) or the data used in the program (to verify the value of the variables used). Clearly, these two kinds of checks can be used to detect security breaches by detecting the corruption of the program execution flow (e.g., code injection) or the corruption of data items (e.g., system call arguments alteration).

An attack can be considered effective once the first illegal system call has been successful. To protect a system call against non-control-data attack, we propose the use of defensive programming to ensure that before a system call is effective, the data items it depends on are in a consistent state at the actual time of the system call.

The data behavior model that we have previously defined gives for each system call the set of variables on which constraints are defined. Building a detection check at the system call is not always possible, as all variables are probably not visible at this point of the program (like local variables located in previous functions in the call stack). Thus, it is necessary to insert checks where the other variables are visible in the program. We have chosen to insert the assertions at each function call in the program, in order to be able to check all the variables when they are visible on the paths of the program that lead to the system call.

### 4. Tool Description

In order to evaluate our approach against non-control-data attacks on real threats, such as the ones described by Chen et al. [7], we have developed a tool that works on source codes
from programs written in C. Our tool generates invariants computed using our approach described in section 3.2 and insert them into the analyzed source codes in order to produce a hardened version of the program. The generation process as well as the insertion process is fully automated and can be easily integrated within any automatic building process.

Our tool is a source to source translator that transforms an untrusted program into a data consistency checking program. It takes a C program as input and outputs an instrumented version of this program that checks, using assertions before calls, invariants computed using our approach.

4.1. Frama-C Description

The invariants that are needed to instrument a program are computed by statically analyzing its source code. To perform this actual analysis, our implementation relies on Frama-C [24]. It is a modular framework dedicated to the analysis of source codes from programs written in C. Among the many features it provides, we are using dependencies and slicing computation using program dependency graph techniques as well as variation domains computation using abstract interpretation techniques [21].

4.2. Plug-in Description

The computation of invariants is implemented within a Frama-C plugin. While parsing the source codes, each time a function call is encountered, the plugin computes an invariant as described in section 3.2. In order to do that, first the plugin use the results of the Program Dependence Graph plugin to compute as described in subsection 3.2 the set of variables the call depends on. Once this set is computed, then the plugin uses the results of the Value Analysis plugin to compute the variation domain of this set.

Within the set of variables, all the variables may not have their variation domain computed. This mainly happens when their value depends on data items that are unknown from the source codes, such as external inputs or results from external libraries. So the plugin selects a subset of variables that can actually have their variation domain computed from all the data the call depends on. Then it computes the variation domain of the tuple containing all the variables from this subset.

For example, using the Value Analysis plugin, we can evaluate for the system call line 6 (see Figure 3) what are the variation domain of the variables a and b. Here, we would find \( a \in \{1, 2\} \) and \( b \in \{0, 1\} \). But this result would be imperfect, as we would have lost the constraints linking a and b. In practice, we know that if \( b = 0 \) then \( a = 1 \) and if \( b = 1 \) then \( a = 2 \). This would lead to a constraint where \((a, b) \in \{(0, 1), (1, 2)\}\). This constraint is more precise that a simple check on the variation domain of the two variables taken separately. In order to be able to obtain such constraints, we must evaluate separately all variables on all possible execution paths leading to the system call. In the example on the Figure 3, we have two paths leading to the system call. While on the first path, \( b = 0 \) and \( a = 1 \). On the second path, \( b = 1 \) and \( a = 2 \).

The Value Analysis plugin does not directly exhibit this result, but it has internally the information required to build these constraints. In particular, a slevel parameter is available in Frama-C that defines the number of paths that can be explored in parallel, and permits the computation of the variation domains on each path. In order to use this information in the Value Analysis plugin, we need to use a hook to access the internal information. The invariant is built by using the variation domain of all variables on each path. In the example we have described, the invariant generated for line 6 is \(((b == 0) \land (a == 1)) \lor ((b == 1) \land (a == 2)))\).

Another limitation of the Value Analysis plugin is that it performs well on pointers, integers and floats, but when it comes to actual data contained by a pointed buffer, a lot of imprecisions are to be expected. It is even more true with strings, since the Value Analysis plugin does not have the specification of the standard string functions. This particular limitation implies that we will not detect attacks on string buffers like the one in the example given Figure 2. For this reason, we have chosen to preprocess source codes to replace standard string comparisons by a set of character comparisons. This way, the analysis performed by Frama-C can be used to compute some constraints on string buffers.

4.3. Plugin use in a real example

Our objective is to show the applicability of our Frama-C plugin in the context of a real application. Several objectives must be fulfilled, such as its insertion in the building process of the application, and the results obtained (that are presented in the next section). In this section, we explain how the plugin has been used to build automatically a security enforced application.

Our tool can be included into the configuration of automatic application building process: the current implementation provides a way to cooperate with an existing standard unix Makefile process without any modification. Thus, a
lot of existing programs written in C and using a Makefile for the building process such as OpenSSH can already be automatically instrumented without requiring any human intervention. Also, since our approach does not require any specific configuration nor any operating system modifications, our tool can easily be deployed on existing systems. Our implementation is specifically targeting C code. To show the feasibility, we use our tool on the OpenSSH building process: the end user just needs to recompile the untrusted program while adding an extra command between the configure process and the make process.

The computation of invariants is the time consuming part in our approach. Therefore, once this step is done, the end user can just recompile the program many times without suffering the cost of the analysis each time. However, since the configure process might influence the way the program is compiled, rerunning this process induces the need to rerun the whole the assertion computation process.

5. Results

In order to evaluate our approach, we must determine if our tool is capable to instrument correctly the sources, and the temporal overhead induced by the detection mechanisms.

5.1. Instrumentation of OpenSSH

Since we have used OpenSSH to illustrate the potential severity of non-control-data attacks, we decided to conduct our experiments on this example. The particular vulnerable version of OpenSSH on which the null password attack can performed [18] has a preprocessed source code of about 38000 lines with 519 functions and 3707 calls. The instrumentation process took 17 hours on an Intel Core 2 running at 2.8Ghz with 8Gb of RAM and generated 291 assertions. This means that almost 8% of the calls have been instrumented. This is an important result, as we can not expect to be able to statically discover the definition domain of all variables inside the program.

5.2. Temporal Performance

We have measured the overhead that our instrumentation induces by scripting the execution of a lot of various shell commands over SSH. The commands represent a wide range of possible shell commands in order to cover the execution of the main ssh server functions. We have performed such measurement numerous times on an instrumented version of the server implemented by OpenSSH and on the original one. The goal was to compare the execution of the two versions. On the average, the overhead induced by our instrumentation is less than 1%.

We can compare our detection model with other methods focusing on non-control-data attacks like Data Space Randomization [13], Data-Flow Integrity [14] or Write-Integrity Testing [15]. These approaches induce an important execution overhead. That is not the case of our approach that induces an unnoticeable overhead.

5.3. Detection Performance

Within the 291 generated assertions, there is an assertion similar to the one presented in the example code in Figure 2, therefore the string buffer attack presented in section 3.1 is detected.

Using this vulnerability, many attacks can be performed. They aim at corrupting various data items in the program (in practice the vulnerability provides a way to corrupt any memory location). The detection mechanisms inserted in the source code would probably permit to detect these types of attacks.

We do not expect our method to detect all attacks on non-control-data: the detection accuracy relies on the completeness of the data behavior model, and thus on the quality of the assertions generated. And nothing ensures that all the assertions are discovered. Thus, we have to expect false negatives. However, when a constraint is broken, we are sure that a program misuse has happened, and thus no false positive can be generated. In practice, we have experienced that in normal executions, no alert is raised.

6. Conclusion and Future Work

This work demonstrates that it is possible to build an intrusion detection model based on data in an application to detect attacks that traditional control flow intrusion detection methods miss. The use of the static analysis framework Frama-C has led to the creation of a tool that instruments software in order to detect non-control data attacks. This tool has proved to be useable in the context of a real software such as OpenSSH. However, the assertions that are generated by our tool only focus on the expected data definition domains. This is clearly a limitation, because it does not permit to detect attacks on variables whose definition domain is unknown in a program. That is why in the future we intend to use additional static analysis techniques that can permit to find relations between these variables. However such methods are expensive in computation time, and we will need to evaluate them in the context of a real project.

7. Acknowledgment

This work was funded by the French National Research Agency (ANR) in the context of the POLUX project. We also thank Pascal Cuq, one of the Frama-C developers, who took the time to explain to us some of the Frama-C’s internals.
References


