Rapport de stage

Master Sciences Technologie Santé

Mention Informatique

Spécialité recherche en Informatique

Université de Rennes 1

2011–2012

Pierre Karpman

École Normale Supérieure de Cachan

Encadré par

Eric Totel & Frédéric Tronel

Supélec

Building up on SIDAN: improved and new invariants for a software hardening Frama-C plugin

Contents

List of Tables

List of Figures

Abstract

We present improvements made on SIDAN, an intrusion detection system working at the software level. The operating principle of SIDAN consists in statically computing invariant properties of the targeted programs and in generating an instrumentation to check those properties at runtime, in order to detect attacks. More precisely, it focuses on invariants involving the values of variables of the program. It checks these invariants when calling functions.

We present improvements on the existing invariants used by SIDAN and propose new invariants as well. We also describe how these have been implemented in SIDAN by using the Frama-C framework, and how they could improve its attack detection capabilities.

Keywords Intrusion detection, software hardening, executable assertions, static analyses, invariants, Frama-C, SIDAN.

1 Introduction

This reports presents a recent work on SIDAN [Dem11], an intrusion detection system that operates at the software level. The objective of such systems is to devise means of detecting attacks on a single, clearly identified software. Typical software that may be protected are Web or SSH servers; these are attractive targets, as they are often reachable from any machine connected to the Internet.

The rationale behind software intrusion detection is trying to prevent the attackers of getting an entry point in a machine (or even in a broader information system) by detecting when the possible entry points themselves are under attack. Complementary approaches consist in trying to detect attacks at a broader level —for instance at the network level. We will however not consider these other approaches in this work.

Several ways of designing a software intrusion detection system are possible. One is to consider the software from an external point of view; a popular approach is to analyse its system call traces and to try detecting attacks from unusual behaviours.

Another common approach is to do *software hardening*; one tries to instrument programs, for instance with a series of checks, in order to detect attacks from inside the software. SIDAN and the systems to which it is the closest belong to this category.

The essence of the hardening performed by SIDAN is to verify invariant properties on the state of the software. Our work consisted in improving the invariants used by SIDAN, and in designing and implementing new ones.

We start this report by exposing the main motivations for software hardening in the context of intrusion detection in section 2. We then describe some works related to ours in section 3, and the approach used in SIDAN in section 4. We talk about more specific points of SIDAN in section 5, where we also expose the contributions of our work in some details. Implementation issues are discussed in section 6, and the evaluation of our system and of

how it compares to the earlier work on SIDAN is described in section 7. We also give a quick tutorial to SIDAN in section 8. We finish this report by outlining a few possible directions for future work in section 9 and by concluding in section 10.

2 Motivation

In this short section, we present general concepts and motivations for a few types of software hardening methods. We show how various sorts of analyses can help to improve the security of a program at runtime.

The section 3 will detail how works related to ours concretely implement these methods.

2.1 Control flow monitoring

When one is interested in software security, and more precisely in detecting attacks on running programs, a quite natural idea is to consider how the running program behaves with respect to its theoretical control flow graph. That is, we would like to detect when a control flow transfer that is not possibly achievable from the source code of the program is occurring.

For instance, if we consider the small C-like program in figure 1:

```
1:   x = 3;
2:   y = 0;
3:   while (x > 0) {
4:       y = y + 1;
5:       x = x - 1;}
```

Figure 1: A tiny program to illustrate control flow monitoring

It is quite clear in figure 1 that the instructions labeled (1, 2, 3, 4, 5) may be executed in this order, whereas an execution of the instructions in order (4, 5, 2, 1, 3) is obviously not possible given the semantics of the source code. If we were to witness the latter order of execution, it would surely indicate that *some problem* occurred at *some point* of the execution (maybe a crazy program counter?). From this observation, we hope that we would be able to detect a part of illegal executions of a software simply by checking that it is following a theoretically possible control flow.

In the specific context of software security —when one wishes to detect intentionally illegal executions (*i.e.*, attacks)— it is quite relevant to monitor the control flow of a program; it is widespread for attacks to inject malicious code in a running software, and then to subvert its control flow to execute the injected code instead of the original[1]. The injected code cannot *a priori* be executed without violating the control flow (*e.g.*, the control flow transfer needed to start running it may not be legal in the first place), and we may therefore hope to detect these sorts of attacks by checking the validity of the control flow at execution.

[1] Categories of such attacks include *buffer overflow*, *return to libc*, and *return-oriented programming* attacks.

Actual methods that monitor the control flow are, *e.g.*, [ABEL05a] and [GRSRV03] (and [KBA02] in some respect), presented in section 3.

2.2 Data flow monitoring

Checking that a program executes along a theoretically possible control flow is a first step towards detecting erroneous executions, but it is not quite enough. It is indeed possible to subvert a program execution without changing its control flow. We illustrate this with the help of the (rather unrealistic) toy program of figure 2.

```
1:   authorized = auth_user();
2:   ask_info_to_user();
3:   if (authorized) {
4:       serve_user();} else {
5:       exit(AUTH_ERROR);}
```

Figure 2: A tiny program to illustrate data flow monitoring

We assume that the variable *authorized* in the program of figure 2 is only modified at the instruction (1). Now if an attacker were able to modify this variable by exploiting a security hole in the function *ask_info_to_user*, he could successfully be served as an authenticated user at the instruction (4), without needing a proper authentication at instruction (1). By doing so, an attacker would still maintain the valid control flow (1, 2, 3, 4) and his attack would therefore pass unnoticed by the control flow monitoring of the previous section.

The objective of data flow monitoring is to make sure that a variable may only be modified from a location in the program where this would be expected, given the source code. In the example of figure 2, such a monitoring would foil the attack we have outlined, as this one needs changing a variable from an illegal location.

Such attacks (and hence protection against them) are relevant in a security context, as a way to exploit security holes is precisely to overwrite arbitrary data in a program (quite often, this is not supposed to happen given the source code), as it is discussed, *e.g.*, in [CXS+05].

In section 3, we present [CCH06], which aims at implementing data flow monitoring.

2.3 Monitoring the value of variables

The last approach we present here is the one that was used in this work. It can be seen as a variation of data flow monitoring in the sense that it also focuses on the integrity of data and ignores control flow issues. The major difference is that it does not consider the data flow, but the value of the data themselves. That is, it does not detect the illegal updates of variables, but the fact that variables hold values that could not possibly have resulted from the source code of the program. For instance, let us consider the program of figure 3

Suppose that the variable *arr* is an array of size 32 and that the variable *i* is not modified in function *ask_info_to_user*; therefore, we can assert that the value of *i* at instruc-

```
1:   for (i = 0; i < 32; i++) {
2:       babar = ask_info_to_user();
3:       arr[i] = babar;}
```

Figure 3: A tiny program to illustrate value monitoring

tion (3) belongs to the interval [0,31]. Now let us imagine that a vulnerability in function *ask_info_to_user* enabled an attacker to modify the value of *i*: this could allow him to perform illegal writes in the memory at instruction (3). However, such a modification could be detected by the very fact that *i* does not hold a theoretically possible value at this point.

The relevance of this approach comes from the fact that concrete attacks, when illegally modifying the value of data, often change them to values that could not be produced by the original code, as this is seen again in [CXS+05].

The works of [Dem11] and [LLHT11], and more generally all of this report follow this approach.

3 Related work

This section expands on the motivations presented in section 2 and presents concrete examples of analyses and of their use to improve software security and safety.

3.1 Control flow methods

Unlike for some of the other approaches that we will discuss in this section, it is rather easy to compute properties on the control flow of a program. The methods that we will overview basically rely on compilation techniques to compute *control flow graphs*.

Hence, the contributions of these methods consist mainly in how to use such graphs to check the validity of a control flow even in the presence of faults or attacks, and to reflect on what security it provides; not in how to compute these graphs in the first place.

We recall that a control flow graph is an oriented graph defining possible transitions between *basic blocks*. A basic block is a sequence of instructions in which no control flow transfer occurs: no execution of the program will lead to a jump to or from the middle of a basic block; any such transfer must occur at the beginning or at the end of a block.

We show an example of a simple program and of its corresponding control flow graph in figure 4 (in this example, the basic blocks are just made of one instruction, and are thus referred to by their instruction number in the graph).

Computing the exact control flow graph of a structured imperative program is always possible when its source code is available (and when it does not use dynamically specified functions) [AP02]. However, this may not be the case when it is computed directly from binary code.

```
1:   if(c1) {
2:       while(c2) {
3:           if(c3)
4:               funk1();
             else
5:               funk2();}}
6:   funk3();
```

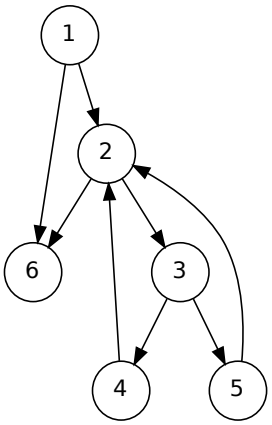

Figure 4: A simple program and its control flow graph

3.1.1 Control flow monitoring for program safety

We now present the works of [GRSRV03] and [VA06]. Although developed to increase program *safety* and not program *security*[2] they are quite similar in spirit to security methods, and we think it is therefore relevant to mention them as well.

There is however a sizable difference between methods for safety and security when the "attack" or "fault" model is considered. In the case of program safety, we tend to assume (this is the case in [GRSRV03] and [VA06]) that faults occurring in the memory are not a concern, because they may be corrected by the use of memory featuring error correcting codes. On the contrary, faults occurring in the registers —and in particular in the *program counter*— are what needs to be detected, and they are the main concern of these methods.

In the case of program security, the "faults" being intentional, error correction is not of any use to protect the memory, and it is therefore considered that it can be arbitrarily modified. Conversely, registers might be assumed to be safe. We will consider the attack models in more details when discussing methods for software security.

The work from [GRSRV03] consists in defining an efficient way to instrument C programs[3] to monitor their control flows. A control flow graph of the target program is computed, and the entry and exit points of its basic blocks are instrumented so as to ensure that:

[2] That is, these methods aim at detecting accidental execution faults, whereas this work is rather concerned with intentional attacks.

[3] The programs' sources need to be available for the approach to be applicable.

1. illegal transitions between blocks are detected (a transition is illegal either if it links to blocks without an edge in the control flow graph, either if it occurs from or towards the middle of a block);
2. the instrumentation has a limited impact on program performance (in terms of execution time and memory consumption).

The instrumentation process uses two phases. The first one is to find the basic blocks and to give each of them a unique identifier; this is done during the compilation of the program. The second phase is to insert the necessary tests to achieve the objectives previously stated. These tests make use of a global variable that plays the role of a witness for the control flow, and that is updated at every new block transition. Then, they simply consist in checking at the start of, say block A, that the previously executed block is indeed a valid predecessor of A; and in checking at the end of the block that we are still executing A.

The second test is useful to detect control flow transitions occurring in the middle of a block. An example of such a transition is illustrated in figure 5. Also in figure 5 however, is an example of a series of illegal transitions that cannot be detected by this method: the checks inserted in [GRSRV03] being only at the limits of blocks: it is not possible to notice if one jumps to the middle of a block, and then back to the middle of the original block.

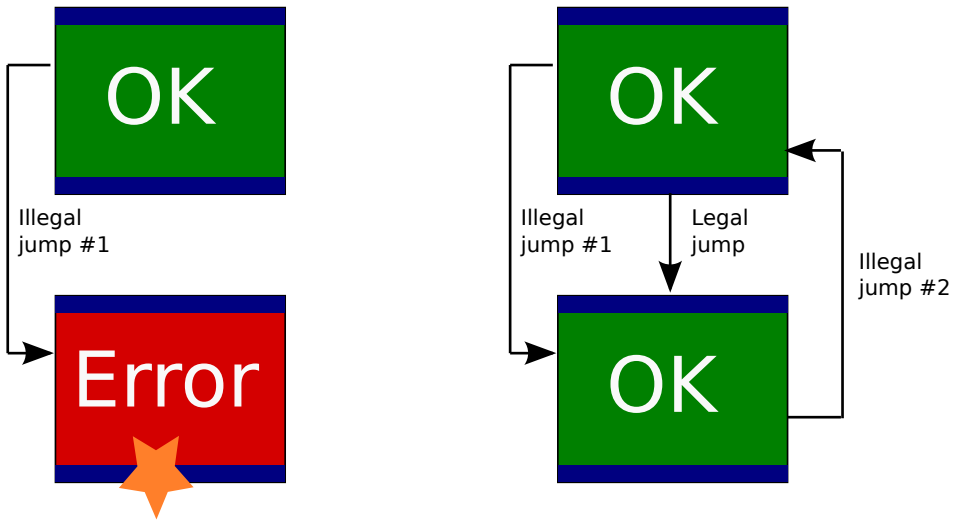

Figure 5: Illegal transitions in the control flow, detected (left) and unnoticed (right)

It should be noticed that the instrumentation must itself resist to faults for the checks to be reliable. We do not detail here how this is achieved, though.

The evaluation of this method by the authors of [GRSRV03] consists in a fault injection campaign, where a small number of programs were executed in an artificial environment creating a large number of faults on the control flow, in a controlled fashion. The results

show that this instrumentation is efficient, detecting approximately 20 to 35% of the injected faults that would otherwise go unnoticed. However, the execution overhead is a 1 to 3.5 multiplier on execution time, and a 1 to 5 multiplier on memory consumption; this is not so reasonable in a context of software security where we expect the overhead to be modest.

The work presented in [VA06] is quite similar from the one of [GRSRV03], as its objectives and techniques are pretty much the same. However, it introduces innovations that make the tests more efficient, hence decreasing the overhead.

The improvements presented in [VA06] come from the introduction of the notion of basic blocks "networks", defined as sets of blocks with common predecessors. The blocks of a same network will all have identifiers which share a common prefix that is computed from the possible transitions between blocks of different networks. Another improvement comes from the idea of distinguishing the blocks with more than one predecessor[4] from the others, and to instrument the two sorts differently.

These improvements permit to decrease the overhead from the one of [GRSRV03], with a detection capacity that remains basically the same.

3.1.2 Control flow monitoring for program security

We now present the works from [ABEL05a] and [ABEL05b], which use the verification of the integrity of the control flow of a program from a security perspective. This approach is very similar to the one of the works of [GRSRV03] and [VA06] from above, as the integrity of the control flow is also verified thanks to assertions inserted at the limit of the basic blocks of the control flow graph.

There is however a range of practical differences between these works. The method from [ABEL05a] does not need the programs' sources to work, as the control flow graphs of the programs are directly computed from executable binaries for *x86* processors[5]. The instrumentation of the basic blocks is also performed by binary rewriting, directly on the executable files. Finally, the assertions themselves need not to be as complex as the ones of [GRSRV03] or [VA06], as the attack model is different.

The result of these differences is a reduced overhead at execution, averaging 20% of time increase. Obviously, it also permits to instrument programs for which the source code is not available.

Yet, the main interest of this work is maybe the formal reasoning on the detection power of this instrumentation, given a well-defined attack model. It also shows how enforcing control flow integrity allows one to efficiently build other components useful for software security. These are the points that we discuss now.

[4] More precisely, we ask from these blocks with multiple predecessors to have at least one predecessor with more than one successor.

[5] The control flow graph that is obtained in this case is a conservative approximation of what the real graph may be, as it is computed from the binary code of the programs (and may thus need to deal with addresses of jumps or function calls that are only dynamically specified, or even worse, with self-modifying programs).

Formal study of instrumentation for control flow integrity The relevance of enforcing control flow integrity and of the methods used to achieve it are rigorously discussed in [ABEL05b]. The authors define a structured operational semantics for a simplified machine code in order to express the normal behaviour of a program and the power of an attacker.

In particular, an attacker is defined as capable of arbitrarily modifying the memory of the data of a running program and the content of a subset of registers. However, he may not change the value of the program counter, the content of the code memory of the program, and the content of a set of distinguished registers (of the number of three). Moreover, it is assumed that the data of a program may not be executed instead of original code (support for non-executable data is now available on some architectures [ABEL05b]).

Under these conditions, the authors prove by induction on the instructions of the simplified machine code that the checks that are placed in the limits of the basic blocks indeed permit to detect violations of the control flow, given a generic formal definition for the control flow graph.

The assumptions made in this work being reasonable, it therefore gives good guarantees on the relevance of the tests they use (which are the same as the ones used in the concrete implementation [ABEL05a]) —even if the proofs were made by using a pseudo-code instead of *x86* machine code.

Constructions built upon control flow integrity A major interest of enforcing control flow integrity is that —by definition— it allows to detect when code that ought to be executed is not. Hence, it ensures that any instrumentation code that is added to a program (*e.g.*, in the form of tests) will either be executed, either trigger an alarm if an attacker successfully bypasses them (because this would mean that he violated the control flow at some point).

As it is explained in the work of [ABEL05a], this allows to perform additional verifications of security-related properties on the programs in conjunction with control flow integrity, thereby improving the whole security of the program.

A first example is the implementation of SMACs (short for *software memory access control*), which allow to build memory zones that are only accessible from specific locations in the program (expressed, *e.g.*, as functions or instructions...). Such a guarantee is interesting for security as it may for instance ensure that hard-coded data in a program and used, *e.g.*, for security verification purposes may not be illegally tampered with. More precisely, the enforcement of control flow integrity eases the implementation of a SMAC by limiting the number of necessary tests thanks to the guaranteed absence of *time of check to time of use* problems[6].

Another example is how to efficiently implement a (P)SCS (short for *(protected) shadow call stack*[7]). This allows to define a security policy on the control flow that is even more precise than the one resulting on control flow integrity enforcement, as it permits to reason on quantitative and qualitative relationships between different function calls (*e.g.*: when

[6] A *time of check to time of use* problem occurs when some data was modified between the moment where it had been checked as valid and the moment where it was actually used.

[7] A *protected shadow call stack* is a copy of the stack that mirrors the real program stack and on which properties may be verified, as it is protected from unauthorized modifications.

function *foo* is called, function *bar* has been called exactly twice as much as function *babar*). The idea of the implementation is again to ensure that the memory segment containing the shadow call stack may only be modified by the part of the code responsible for its update.

3.1.3 Control flow monitoring (and various policy checking) with program shepherding

We now turn our attention to the work from [KBA02], which presents a system designed to supervise the execution of a program and in other things to constraint its control flow so that it abides by configurable security policies.

This system is not unlike the previous works we presented, but its concrete implementation is quite different as it is based on efficient code emulation, rather than direct program instrumentation (at the source or binary level). It should also be noted that it does not provide as strict guarantees on the integrity of the control flow as does the work of, *e.g.*, [ABEL05a]; this comes in part from the fact that it entrusts the user with defining the security policies himself, and therefore does not systematically check that a control flow transfer is valid according to the control flow graph (which by the way is not computed in this method).

This method also offers more than strict control flow checking, as it allows limited reasoning on the values of the arguments of system calls (as we will see shortly). Basically, the objectives of this work are to define and implement three techniques that may then be used to express custom security policies. These three building blocks are:

1. restricting code execution in function of its origin by, *e.g.*, deciding if dynamically loaded or dynamically generated code may be executed;
2. restricting control flow transfer, like for instance limiting the calls to library functions from specific locations in the code, or simply ensuring that a function call returns right after when it was called;
3. inserting executable assertions that may not be circumvented. This building block is in fact used to implement the two previous ones, and more generally it is useful for any security instrumentation that relies on executable tests.

The system presented in this work permits to define generic policies (like, *e.g.*, stating that an indirect function call[8] may only target read-only memory) and policies specific to a given function or system call as well.

For instance, a user may wish to define a policy forbidding the *exec* system call to take any argument that is not hard-coded, or to forbid *open* to allow writing to sensitive files specified in the policy (*e.g.*, */etc/passwd*). Thus, even if an attacker successfully subverts a program running with administrator rights, he may still be unable to actually modify sensitive files or execute arbitrary code on the machine.

The implementation of these building blocks is done by executing the unmodified pro-

[8] A function call is indirect when the identity of the function is not known at the time of compilation.

grams through a "dynamic optimizer"[9], and to dynamically perform the tests dictated by the security policy.

It is always possible to run the programs without checking any security policy whatsoever (*i.e.*, running the program through the bare dynamic optimizer), which means that this method could quite neatly be integrated in a general context, for instance at the level of the operating system.

As quite everything in the implementation is dynamic, the security policies could be defined and updated cheaply, all along the life of the system. This is more flexible than the other methods presented in this report, that need the programs to be recompiled or statically analysed to be instrumented. Finally, when one considers performance overhead, the authors of [KBA02] notice an average time and memory overhead of 10%, although it may be close to 100% in some cases.

The authors do not define a precise attack model as do the ones of [ABEL05a], against which their implementation is proved to resist. However, they do provide a few arguments to explain why their system in itself may not be attacked (hence ensuring that the security policies defined by the user are actually enforced).

The first argument is that the control flow monitoring features of the dynamic optimizer on which they base their implementation guarantee by construction that non-circumventable tests are really possible. The second argument is that protecting the data used by the system (for instance to store the security policies) can be done by a combination of defining appropriate read & write rights for the pages where these data are located, and of non-circumventable checks that make sure that these rights are not illegally modified.

3.1.4 Limits of control flow monitoring

As this was discussed earlier in section 2, it is possible to perform an attack on a program even when its control flow is not altered. This was shown in the work of [CXS+05] that presented concrete attacks on real-life software (like, *e.g.*, the OpenSSH server) that did not need modifications of the control flow. This is part of the motivation for researching new techniques that would allow to detect other attacks than the ones modifying the control flow of a program.

However, one should not hastily conclude that enforcing control flow integrity is useless as it still permits to detect a lot of potential attacks. Moreover, as it was already outlined several times, these methods are very useful as building blocks for implementing other security mechanisms.

3.2 Data flow methods

We will now present methods based on properties of the data flow of a program.

[9] A dynamic optimizer tries to (efficiently) run unmodified native code in order to find opportunities to improve its performance. It may also allow to profile and to dynamically instrument the code, which is what we are most interested in in our context.

3.2.1 Data flow integrity for intrusion detection

We present the work of [CCH06] which aims at preserving the data flow integrity of a program by inserting suitable tests in the source code. What is called *data flow integrity* (or integrity of the *data flow graph*) in this case is based on *reaching definition* analyses, which come from the compilers field [AP02]. A reaching analysis defines the notion of *define* and *use* relationships between variables (or memory locations) and instructions in the program.

In more details, one says that an instruction writing in a variable (or more generally in a memory location), say variable *babar*, *defines* the value of *babar*, and an instruction reading its content *uses* the value of *babar*. What the reaching definition analysis does is computing a superset of the instructions that may define the value of a variable for each instruction where it is used. It is a superset and not the exact set of such instructions because, like many static analyses, this one is approximate [AP02].

However, the fact that it includes all the real instructions means that it is sound. This soundness is interesting as it permits to build a detection system with no false positives; even if false negatives may still occur due to the imprecision of the analysis. We give an imaginary example of what this analysis may compute with the help of the program in figure 6.

```
1:   x = 0;
2:   for (i = 0; i < 30; i++) {
3:       x = x + 2 * i;
4:       if (x % 2) {
5:           x = x + 1;}}
```

Figure 6: A tiny program to illustrate reaching definition analysis

In the program of figure 6, the instructions that may define the value of variable x when it is used at instruction (3) are instruction (1) and instruction (3). The soundness of the reaching definition analysis guarantees that both instructions will belong to the set it computes. However, the analysis being imprecise, one could imagine[10] that it is not able to determine that instruction (5) may never define the value of x at instruction (3), and thus include it in the set as well.

The idea used in the work of [CCH06] is to ensure at runtime that when a variable is used, it was last modified from a point in the program that may theoretically do so, according to its data flow graph. If this is not the case, it necessarily means that an error occurred at some point.

This idea is implemented by the authors of [CCH06] by using a compilation framework for C programs. This framework allows both to implement the reaching analysis and to generate the instrumentation that enforces data flow integrity. This instrumentation consists in keeping a big table indexed by memory location that is updated each time there is a write to

[10] We did not actually perform a reaching definition analysis in this case, so we only make this hypothesis for illustration purposes.

the memory, and that checks the integrity of the data flow graph when a read is performed. This is shown in an abstract way in figure 7.

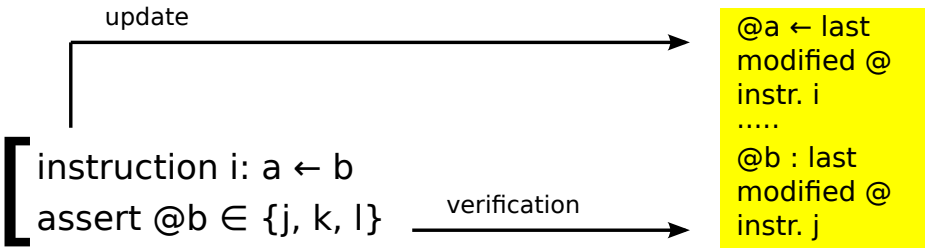

Figure 7: Enforcing data flow integrity around a read & write instruction

A series of optimizations is also described by the authors to make the instrumentation more efficient, like for instance keeping only one entry in the table for equivalent variables (ones with the same superset of possible locations where they may be defined) or avoiding to check the integrity of variables that are known to be inside registers (which are supposed to be out of reach of an attacker).

For this implementation itself to be safe against attacks, it is necessary to ensure that neither the instrumentation code can be avoided, neither the table on which it relies can illegally be modified. This however can be ensured for instance by ensuring control flow integrity thanks to the method of [ABEL05a] and by placing the table inside a SMAC, using the same method. This assumption is therefore entirely legitimate.

It should be noted that as this method needs to insert checks anytime a write is performed, this makes it a bit tricky to deal with library functions which may not be possible to instrument to perform the updates in the table. One way to circumvent this problem is to globally state that any part of the memory accessible by a library function was updated illegally, each time it is called[11]. Therefore, if these parts of the memory include actual locations of variables used in the program, an illegal write performed *via* a library function will still be detected the next time the variable is read.

From a performance point of view, the implementation of this method given in [CCH06] leads to an augmentation of the code size of 50% and to an execution time increased by 20 to 250%. These figures come from an evaluation that was performed on a small set of programs from the SPEC2000 benchmark suit. The authors note that as these programs are CPU-intensive, one could hope that the time overhead would be less important for programs that would more likely be the target of security improvements like, *e.g.*, Web servers. An additional experiment by the authors tend to validate this hypothesis, showing that the time overhead for such a server is negligible when its load is low, and that it is around 20% when it is reasonably important.

[11] With the exception of memory locations accessible thanks to a pointer given to these functions as argument.

3.2.2 Data flow external monitoring

The work from the previous section is not the only approach that makes use of a notion of data flow. For instance, the work of [BCS06] discusses a method to compute fine-grained relationships between the arguments of the different system calls of a program, by observing its system call trace. This allows to build a model of the normal behaviour of a program to be later used to detect attacks that make the program drift from this model.

However, the methods used to achieve these objectives are very different from the others we considered so far, as they rely more on an external, statistical approach rather than on source or binary analyses. We will therefore not consider them further in this report.

3.3 Data value methods

We close this section by discussing methods which focus on the theoretically possible variation domain of programs' variables. This approach is quite important in this report, as our work precisely consisted in improving the work of [Dem11], which is an approach falling into this category. Due to its importance regarding our work, this method will be described in a section of its own, in section 4. We will however quickly describe a similar approach in the remainder of this section.

We present the work of [LLHT11], which aims to detect attacks thanks to how they modify the value of variables of the program, even if this does not lead to a violation of, *e.g.*, its control flow. Unlike most of the other methods presented in this section, this system focuses on software written in Ruby (and not in C), thereby targeting Web applications.

The idea used in this work is to compute invariants on the value of variables and then to check that these invariants hold when the program is executed. An alert is triggered when this is not the case. In practice, two issues must mainly be addressed: choosing the variables on which computing the invariants, and actually computing invariants.

The choice of variables made by the authors of [LLHT11] is to consider the variables that are influenced by user inputs. These inputs are the entry point of any potential attack, and therefore one could argue that an attacker needs to subvert the value of variables that may be influenced by them in order to perform his attack successfully. In practice, this set of variables is determined by using a method of data tainting[12]. This can be implemented efficiently in Ruby, as it already features a tainting mechanism.

When it comes to invariants, the implementation of [LLHT11] uses heuristic invariants produced by the Daikon invariant generator [PAG]. Daikon tries to generate invariants from execution traces collected from runs of the program to be instrumented, and may be customized in order to try generating invariants of a form specified by the user. In the case of [LLHT11], the invariants it tries to find are: whether a variable is constant or takes its value from a finite set of elements; whether two variables are equal; and whether two variables are ordered in a certain way by an order relationship.

[12] That is, runs of the program are performed that *taint* a variable whenever it is influenced (directly or not) by a user input

A preliminary evaluation from the authors of [LLHT11] shows that an instrumented program may indeed be able to detect attacks that illegally change the value of data of the programs, by the way of, *e.g.*, SQL injections. However, no estimation of the performance overhead is given.

Let us conclude by saying that this is yet another system that relies on tests that must themselves be protected against attacks; but this can be provided by external means (even if there may be quite fewer of them when the programs are written in Ruby instead of C).

4 Our approach

This section presents some generic aspects of SIDAN [Dem11], which is the technique studied and improved upon in this work. We first give an overview of the approach and explain some design choices, then we describe in more details (compared to what has been done so far) the technique of executable assertions, and we finish by a presentation of our attack model.

4.1 A data value method

SIDAN is a system that exploits the value of (important) data of the programs in order to detect attacks, something that was already presented in section 2 and section 3. The choices made in SIDAN are however quite different from the ones of [LLHT11] —the technique of section 3 with a similar approach.

SIDAN targets C programs for which the source code is available, and is implemented in the Frama-C framework [CEAb].

We recall that a data value method tries to find invariant properties on the value of data of the programs, and then checks at runtime that the invariants hold. A practical implementation of such a method faces two main choices: deciding on which variables computing the invariants, and what sort of invariants to compute (and how). We now describe those choices.

Choosing relevant variables The choice of variables of SIDAN is based on the idea that an attacker will most certainly try to influence the execution of system calls, because it is the only way for his attack to have a direct impact on the system where the software is executing[13]. Thus, the variables chosen in SIDAN are the arguments of system calls (and actually also any function call), with the addition of the variables that influence the execution of the calls (*e.g.*, test conditions or loop counters).

Along with the choice of variables, one ought to choose where in the code the invariants properties of these variables are computed (and hence where instrumentation will be inserted). The answer of this question may not be as immediate as for control and data flow methods, where the locations of the tests is a direct consequence of the nature of what is

[13] By making this assumption, we implicitly consider that we are not so much focused on detecting *denial of service* attacks, as those do not necessarily need to directly influence their environment.

computed and of the attack model. In the case of SIDAN, the choice made by the authors is to compute the invariants around the system and function calls (which may seem natural as those are already the focus of the variable choice).

In practice, if the set of arguments of a function is easy to find, it may not necessarily be so for the set of variables that influence the call. This can still be done, though, by using *program slicing* techniques [Wei81], and Frama-C features a ready-to-use such analysis — roughly speaking, a program slice is a sub-program computed around an instruction, which behaves the same way as the whole program as far as only the said instruction is considered. However, one should still note that this analysis is approximate, and it may yield a superset of the variables that may really influence a call.

Computing invariants The various invariants that are computed on the target variables are described in great detail in section 5. Here, we only mention the fact that from a generic point of view, these invariants are computed statically thanks to abstract interpretation techniques [CC77], using the analyses of Frama-C.

Performance From a performance point of view, SIDAN has a quite low impact on the execution time and memory consumption of the instrumented programs. This is because it only adds tests around the function calls, which are not the most prevalent instructions in programs. For the same reason, the augmentation of the code size is quite limited. A series of fault injections was used to estimate the relevance of this method, and showed that approximately 70% of the faults injected in the tested SSH server were detected. However, one should note that the static analyses used to generate the invariants are themselves quite costly, and therefore it may take a long time (up to several hours) to instrument a program. Performance issues of SIDAN are further discussed in section 7.

4.2 Using executable assertions for invariant checking

This section describes in some detail the rationale for using assertions in SIDAN, and how they are structured.

4.2.1 Wait... why do we want this?

SIDAN is based on the computation of invariants on the value of data of the program. Once these invariants have been computed, it is necessary to devise a way to use them as means of intrusion detection. The approach used in SIDAN is simple, and consists in using executable assertions inserted throughout the source code. This is similar to what the related works of section 3 did. In the case of SIDAN, the goal of these assertions is to check that some properties (*i.e.*, the desired invariants) are fulfilled at some specific points of the program (*i.e.*, the function calls).

The nature of the invariants in SIDAN is quite amenable to verification by an assertion. As the invariants express properties on the values taken by programs' variables, these prop-

erties can be translated in a programming language in a straightforward fashion. Therefore, the whole "power" of an invariant can fully be captured by an assertion.

The fact that the assertions we insert are executable is merely to make the intrusion detection practical; it is entirely possible to imagine non-executable assertions inserted, *e.g.*, in the form of comments. This could still be useful for informative purposes or making proofs of programs. As we aim for the programs to detect violations of their invariants at runtime, though, making the assertions executable is much more relevant in our case.

Even though we say the assertions are executable, there is no real need to specify at this point what the effect of a failed assertion should be[14]. We think that this choice is really up to the end-users of an intrusion detection system, and we thus only need to let the effect of a failed assertion to be easily configurable.

4.2.2 Structure of the assertions

We now detail the different structures used to build the assertions in SIDAN; these will depend slightly on the sorts of invariant that are computed, and three different types of assertions may be used.

1. The first type is an assertion that is inserted just before a function call, and that may involve any variable currently in the scope of the program.
2. The second type is identical but for the fact that it is inserted right after the function call.
3. The third and last assertion type is made of two parts: one part inserted before the function call, consisting in the declaration and assignment of local variables, and a second part inserted right after the function call and being the assertion proper; in this case the assertion may also involve any variable currently in the scope, including the ones that have been declared just before the function call.

It may seem mundane to distinguish between these different sorts of assertions, but their differences in fact lead to important implementation issues, as it will be discussed in section 6. The three sorts of assertions are described in figure 8, where P is an unspecified predicate responsible for checking the validity of the invariants.

Type 1: `assert(P(var_in_scope1, var_in_scope2, ...)); funk(args);`
Type 2: `funk(args); assert(P(var_in_scope1, var_in_scope3, ...));`
Type 3: `new_var = val1; ... funk(args); assert(P(new_var, var_in_scope1, ...);`

Figure 8: The three possible structures for an assertion

4.3 Attack model

In this section we explicitly state the hypotheses we make about the abilities of the attackers whose attacks we wish to foil. This will enable us to precisely know when it is (not) possible

[14] Although we can still suggest a few possibilities like writing a log entry or terminating the program (either in an orderly fashion or not)

to circumvent our system, and also what sort of attacks are detected when the system is effective.

This also provides guidance to an end-user, so that he would be able to take all the necessary measures to make an instrumentation by SIDAN meaningful.

4.3.1 Non-modification of the code

We suppose that the attacker is unable to dynamically change the code of the program we wish to protect. If this were not the case, avoiding our assertions would simply be a matter of removing or rewriting them so as to prevent any alarm from being raised.

This assumption is not valid in a completely generic setting but is still quite reasonable when we suppose that a user of SIDAN is serious about improving the security of the programs he instruments. In this case, it would be logical for him to also execute the program in a hardened environment —like one provided by, *e.g.*, program shepherding [KBA02]— and therefore to express security policies that forbid the dynamic modification of code.

4.3.2 No direct modification of the content of the registers

We make the assumption that is is not "realistic" for an attacker to be able to change the content of the registers inside the processor.

This assumption is necessary, because an attacker able to do this could then pass through our assertions unnoticed by directly changing the values against which the variables are checked. For instance, suppose we check that the variable *var* is either 0 or 1, and that an attacker needs to change *var* to 29 for an attack to succeed; if he is able to modify the content of the registers, he could change the assertion so that it checks that *var* is either 0 or 29, completely invalidating the protection offered by the assertion.

Similar assumptions are made, *e.g.*, in [ABEL05a], where it is assumed that an attacker cannot modify a few distinguished registers at some points of the execution of the program.

4.3.3 Impossibility to bypass the executable assertions

We also make the quite natural assumption that it is not possible to bypass our assertions. This is obviously necessary for our instrumentation to be of any use.

Although this assumption is far from being true in a general context, it is perfectly valid if one uses SIDAN in conjunction with a system that preserves the integrity of the control flow of a program like, *e.g.*, [ABEL05a], or one that allows the definition of non-circumventable tests like, *e.g.*, [KBA02].

4.3.4 Possible modification of the content of the memory and data segments

What SIDAN may protect against, then, is the modification of the content of the memory of a running program or of its data segments to alter the value of a variable. In this case, if the new value of the variable is such that it violates one of the invariants computed by SIDAN, the attack will be detected.

5 The different sorts of invariants

In this section, we present the different sorts of invariants that we compute with our technique. The difference between an invariant and an assertion that checks its validity being quite small, we choose to present the invariants in the form of their corresponding assertions. This choice is motivated by the fact that it helps to highlight the difference between a "theoretical" invariant and what we are actually able to compute with a concrete implementation of our technique. Indeed, some of the contributions of our work are directly related to implementation issues and consisted precisely in creating new *assertions* for *invariants* that were already known[15].

As an example, we show in figure 9 the notation that we will use for the call of an imaginary function "funk" of arguments arg_1 and arg_2, saving its result in *res*; the notation for a corresponding imaginary invariant on the variation domain of arg_1 and arg_2; and the function call instrumented with the assertion corresponding to the invariant.

Bare function call:	$\{res \leftarrowtail \text{funk}(arg_1, arg_2);\}$
Invariants before the function call:	$arg_1 \in \{0,1\} \wedge arg_2 > 0$
Instrumented function call:	$\{\text{assert}((arg_1 = 0 \vee arg_1 = 1) \wedge (arg_2 > 0));$
	$res \leftarrowtail \text{funk}(arg_1, arg_2);\}$

Figure 9: Notation for code excerpts, invariants, and executable assertions

5.1 Original invariants

We first present the invariants as originally computed in [Dem11]. These invariants, for a given function call, express constraints on the variation domain of as many variables as possible, provided they are in the current program scope and in the program slice computed around the said function call.

Given that we want to protect the program against malicious function and system calls, the variation domains that are considered for the invariants are the ones right *before* the call. One could consider the alternative of checking the invariants computed *after* the calls; the main consequence this choice would have is that the detection of a malicious call would become possible only after it occurred, under the additional condition that the function returns[16]. The consequence of all of this is that the nature of these invariants leads to assertions of the first type described in section 4.2.

The actual invariants used in SIDAN closely reflect the capabilities and limitations of the system used to implement them, *i.e.*, Frama-C. What we describe now is therefore the original invariants such as they are computed in practice.

[15] The implementation part of this work is thoroughly discussed in section 6.

[16] Of course the invariants before and after the call have no reason to be the same in general, and therefore the choice between the two is real.

5.1.1 Basic invariants on integer types

The first type of invariants computed in [Dem11] leads to assertions on the program's integer variables (as we consider C programs this means, *e.g.*, variables of type *int*, *unsigned int*, *char*...), checking the simultaneous membership of each variable to an explicitly defined set. This is illustrated by the following example, where a function call is instrumented with tests to check that the variables arg_1 and arg_2 both belong to the set $\{0,1\}$:

$$\{\text{assert}((arg_1 = 0 \vee arg_1 = 1) \wedge (arg_2 = 0 \vee arg_2 = 1));\ \text{funk}(arg_1, arg_2);\}.$$

An implementation limitation of those invariants is that they may not involve arrays or structured variables.

5.1.2 Aggregated invariants on integer types

One nice evolution of the previous invariants presented in [Dem11] is to take into account the fact that not all of the elements of the Cartesian product of the possible variation sets for each variable are possible. In other words, it may be that the value taken by one variable restricts the possible values for another variable to only a subset of all of the values it can otherwise assume. For instance, if we consider the previous example where arg_1 and arg_2 both belong to $\{0,1\}$, one could imagine that when $arg_1 = 0$, then we necessarily have that $arg_2 = 0$.

What was implemented in [Dem11] is a way to obtain invariants that only consider the configurations of values that are found to be possible. Therefore, if we consider the previous example, with the constraint that $arg_1 = arg_2$, the corresponding instrumentation of the function call would be:

$$\{\text{assert}((arg_1 = 0 \wedge arg_2 = 0) \vee (arg_1 = 1 \wedge arg_2 = 1));\ \text{funk}(arg_1, arg_2);\}.$$

Of course, such relations between the values of variables may not exist, and even when they exist it may not be possible to actually find them. What we say in this paragraph is therefore just that when it is known, such an information will not be ignored and will make its way to the actually inserted assertion.

It should be noted that despite the appearances, the invariants that are computed here are not the results of analyses computed on so-called "relational domains" (described, *e.g.*, in [Min04]), which would enable one to express properties such as $0 \leq arg_1 + arg_2 \leq 1$; even if those may sometimes give results equivalent to what is computed in SIDAN.

5.2 Invariants on arrays and structured variables

We now describe the first improvement we realised on the original invariants. This improvement does not change the type of the invariants, which means that the assertions they produce will still be inserted before the function calls, and will still involve variables from the same program slice as before. The only difference, in fact, is that the invariants may now express constraints on C arrays and C structures, thereby increasing the possible number of involved variables. As an example, the following assertion could now be generated:

$$\{\text{assert}((arg_1 = 0 \vee arg_1 = 1) \wedge (arr_1[0] = 2) \wedge (stru_1.\text{a}[2] = 0));\ \text{funk}(arg_1);\}.$$

It is however important to note that these invariants still involve variables that are akin to integers; given an array or a structure, it is still not possible to express invariants in terms of addresses. For instance, for a variable arr_1 declared as *int *arr1[64]*, it is currently not possible with our system to produce invariants of the form $arr_1[7] = @_7$, with $@_7$ being some address in the memory[17]; similarly, producing the invariant $stru_1.\text{a} = @$ for a structure $stru_1$ which field "a" is of type *int ** is currently impossible. The reasons for this improvement come mainly from changes in the implementation.

5.3 Interval approximations for the variation domains

5.3.1 The return of the variation domains

So far, the analyses used to compute the invariants in SIDAN reasoned in terms of variation domains defined as explicit sets of integers; that is, it reasoned on properties of the form $var_1 \in \mathscr{S}$, with $\mathscr{S}$ an explicitly defined set of integers such as $\{0,1,2,7,1200,4520\}$. When such an invariant needs to be expressed as an assertion, all there is to do is creating an expression of the form:

$$\bigvee_{v \in \mathscr{S}} var_1 = v \tag{1}$$

When the tweak from section 5.1 is used to obtain aggregated invariants, the variation domains are defined in a slightly different fashion, as properties of the form $\langle var_{1..n} \rangle \in \mathscr{T}$ with $\mathscr{T}$ an explicitly defined set of elements of $\mathbb{Z}^n$ such as $\{\langle 001 \rangle, \langle 221 \rangle\}$. Expressing such an invariant as an assertion is straightforward, leading to expressions of the form:

$$\bigvee_{u \in \mathscr{T}} \bigwedge_{i=1}^{n} var_i = u_i \tag{2}$$

Expressing the variation domains in this way is conceptually easy, but it is not without limitations. There are mainly two of them, which are in fact closely related:

- firstly, if the set $\mathscr{S}$ (resp. $\mathscr{T}$) contains more than a few elements, converting a variation domain into an expression of the form 1 (resp. 2) might be inconvenient, in the sense that the resulting expression would be difficult to read and lengthy to check;
- secondly, it is inconvenient in itself (*i.e.*, expensive in terms of computing power and memory) to produce big sets as the results of value analyses, as it implies the handling of big objects all the way through the value analysis algorithms.

All this means that this approach does not scale well to big variation domains, although it is practical and pretty relevant when working on small domains. An alternative way of representing the variation domains seems therefore necessary.

[17] It was already —and it remains— impossible to obtain invariants of the form $var_1 = @$ for some variable var_1 declared, *e.g.*, as *int *var1*.

5.3.2 How Frama-C does it

The default behaviour of the value analysis of Frama-C is to define the variation domains of variables as sets up to a certain point, and then to approximate the sets by intervals of integers when they become "too big". For instance, the variation domain for a variable that may assume the values 1, 2 or 3 is defined as $\{1,2,3\}$. However, if the same variable could assume any integer value between -500 and 1230, its variation domain might be defined as $[-500,1230]$, which is obviously more convenient a result to handle than a set of almost 2000 elements.

Using intervals to describe variation domains is of course not without its own drawbacks, as it often leads to a loss of precision. How important this loss may be is of course highly dependent of the context; approximating the variation domain of a variable that may assume the value of any prime number between 0 and 100 000 by the interval $[2,100000]$ is of course imprecise, whereas no loss of precision whatsoever occurred in the former example from above.

One way to mitigate the loss of precision led by the use of intervals is to express additional constraints for elements to be part of the variation domains. In Frama-C, this is implemented in the form of one congruence relation that can potentially be added to the original interval. For instance, if a variable var_1 may assume any value from a big set bounded by 0 and 2000 (say too big to be handled explicitly as a set), and if it happens that it never assumes an odd value, then what we will obtain from the value analysis plugin of Frama-C is that $var_1 \in [0,2000] \wedge var_1 \equiv 0[2]$. However, we stress again the fact that the plugin does not use relational variation domains, and thus does not produce invariants such as $0 \leq var_1 + var_2 \leq 1$.

5.3.3 How does SIDAN use what Frama-C does

As it is said at the beginning of this section, SIDAN so far only considered variation domains expressed as sets. A motivation for this choice is that using interval domains is not compatible with the advanced "aggregated invariants" presented in section 5.1; using these implies toggling off the interval approximations of Frama-C, thereby preventing any loss of precision at the cost of what is just mentioned above.

Even if this choice is perfectly sound, we think that it may be of some interests to use invariants on interval domains as well, as a compromise proposed to the end-user. In this case, when aggregated invariants are *not* used, the invariants in SIDAN may use both types of variation domains. This reflexion led to a change in the implementation of SIDAN in order to take interval domains into account, now creating an assertion in the event of an invariant using such a domain. Constructing the assertions is pretty straightforward, the invariant $var_1 \in [b,t] \wedge var_1 \equiv r[d]$ leading to an expression of the form:

$$b \leq var_1 \wedge var_1 \leq t \wedge var_1 \equiv r[d] \tag{3}$$

5.4 Invariants on the return values of functions

The original invariants presented in section 5.1 are computed *before* the function calls; we now present a new sort of invariants computed just after the functions return. The idea of this invariant is very simple, and simply consists in checking for a function that its return value is indeed within its theoretically possible variation domain. For the same reasons as for the previous invariants, computing these is only possible when the functions return an integer value, thus excluding functions that return pointers.

It is obvious that the instrumentation used for these invariants will be of the second sort as described in section 4.2, with checks being inserted right after the function call.

The rationale behind this sort of invariants is to make amend for the fact that the value returned by a function is not part of the original invariants; this would have been possible if those had been computed after the function returns.

5.5 Invariants on locally constant variables

5.5.1 Original idea for new invariants

We now present the last sort of new invariants that we introduce with this work. The initial motivation for these invariants was to enhance the original ones by making them become inter-procedural, by way of constraints propagation. In more details the idea was, given a function and its corresponding instrumentation, to determine which of the variables used in the instrumentation would be bound to the same constraints when present *inside* the said function. Once these variables are found, their constraints would be propagated down to every function call (if any) in the body of the original function.

This can be seen as an opportunistic extension of the original invariants: one tries to check the variation domains of variables as much as possible, even at points where these variables would not be expected to be used. This idea is illustrated in figure 10.

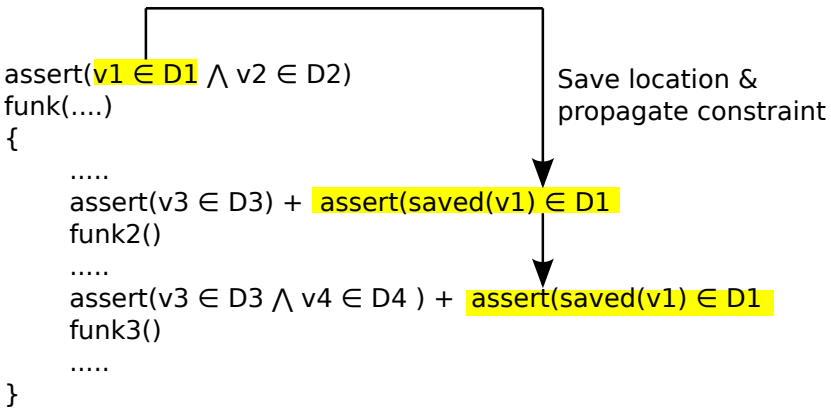

Figure 10: Propagation of the constraints on a variable

Implementing this idea is not without hurdles, and mainly two issues would need to be addressed: the first is finding a way to determine which variables should have their constraints propagated, and the second (quite more problematic) is to devise a method to actually check these constraints, that may involve variables not in the scope at the moment where we wish to check them.

5.5.2 Think different

A simple observation actually allows a rather efficient implementation of a very similar kind of invariants in the case of terminating functions. The idea is to consider those of the variables from the original invariant that are left unmodified by the "real" code of the function, and to ensure that they were indeed left constant once the function returned.

This can be easily implemented by saving the value of the relevant variables into temporary variables, just before the function call, and then by checking the variables against their saved value just when the call finished[18]. In other words, the assertions built upon those invariants are of the type 3, as shown in the figure 8 of section 4.2.

The only remaining obstacle to this analysis is thus to find precisely which variables are not modified by the function.

It should be noticed that the constraints that are verified by these invariants may be quite strong, in particular allowing to check properties on addresses (typically on the value of pointers and on the base value of arrays) which is something that —as already said— is not deducible from the results of the value analysis as it is currently used.

We can also see that when combined with the original invariants, this technique also makes redundant any check on the variation domains after the call (as previously discussed in section 5.1) for the variables that remain unchanged after the function returns and for which the variation domain was known.

This last point can be illustrated with the following example: consider a function "funk" that does not modify the value of a variable var_1, which variation domain before (and thus after) the call is the interval $[0,100]$. By saving the value of var_1 and verifying that it is the same after the call, it is not only possible to check that it is still in the interval $[0,100]$, but also that its value did not change to another one (in the interval or not), for instance detecting if var_1 changed, say, from 29 to 37. The two instrumentations are compared in figure 11.

The difference between this approach and the one of the previous section is that the assertions are only checked when the function returns, which means that we may lose invariants that could have been computed on non-terminating functions. On the other hand, the invariants that we compute are more precise than what could have been obtained by constraints propagation, as we check the variables against a single value.

[18] It is quite obvious that in this case, our attack models relies again strongly on the fact that the content of registers may not be modified by an attacker, as we expect the integrity of the saved values to be preserved

```
assert(0 <= var1 && var1 <= 100);
funk(var1);
assert(0 <= var1 && var1 <= 100);
```

```
babar ← var1;
assert(0 <= var1 && var1 <= 100);
funk(var1);
assert(babar = var1);
```

Figure 11: Assertions on variation domains after the calls and of constant variables, compared.

5.5.3 Finding locally constant variables

It is quite fortunate that Frama-C already implements an analysis that fits our objective of determining which variables are not modified by function calls. This analysis is provided by the *scope* plugin (which itself relies on the value analysis), and enables us to easily compute the information we seek.

The candidate variables that are used for this invariant when they are determined to be constant are taken from the program slice computed around the function call, pretty much in the same way as the variables involved in the other sorts of invariants.

Of course, one variable may be involved in one sort of invariant and not in another, and it is often the case in practice that one variable is found to be constant even if its value is not known at all.

5.5.4 Address verification

We now would like to highlight the use of this new invariant in the case of the verification of the values of pointers and of the base addresses of arrays.

It is quite obvious that an attacker able to tamper with the value of one such variable will then be capable of changing the value of whatever part of the memory that can legally be accessed from the attacked variable[19]. Being able to detect pointer tampering is therefore an important part of attack detection in general.

One should also note that in the case of arrays (or similarly, in the (maybe less common) case of pointers on which pointer arithmetic is performed), arbitrary writes in the memory can be achieved in two ways: the first by tampering with the indices used for the array (in which case the currently discussed invariant is useless; but this attack implies using values for the indices that lie outside of their expected variation domain, and therefore it may be detected by the original invariant); the second by tampering with the base value of the array, so that the regular accesses to the array in fact lead to the desired memory location.

In the case of the invariant presented in this section, it is pretty straightforward to create the assertions needed to ensure the non-modification of pointers within a function call, as pointers are seen like any other variable by the scope analysis.

[19] Here, legally means, *e.g.*, in the same segment of memory, so that the access will not raise a runtime error.

5.6 Summary of the progress on invariants

In this section we briefly go through the advances that have been realized on the invariants of SIDAN that have just been described:

1. The invariants may now involve integer arrays and structures, and their variation domain may be expressed in the guise of an interval (5.2, 5.3).
2. A new invariant makes use of the variation domain of the value returned by a function, hence leading to assertions inserted after the function call (5.4).
3. A new invariant expresses the fact that some variables are not modified by a function call, hence leading to assertions inserted after the function call, using temporary variables introduced right before it (5.5).

The impact of these improvements is evaluated in section 7.

6 Implementation

This section presents details about how all of what has been described so far is actually implemented. Implementation was the main focus of this work, and it is therefore only fair to dedicate a large amount of this report to implementation-specific points. We start by presenting Frama-C [CEAb], on which our implementation is based, then we give an overview of our work, and we end by detailing various implementation choices that were made to answer concrete challenges and issues.

6.1 Frama-C

This section is dedicated to the presentation of Frama-C and of how it interacts with our work.

6.1.1 Frama-C overview

Frama-C is a *framework for the modular analysis of C programs* developed by the CEA [CEAb]. As the name implies, Frama-C is not a single, heavyweight software that invariably serves a single purpose (whatever sophisticated that purpose may be), but rather a collection of tools made available to the user; the said tools may be used independently (although some may depend on others) or in conjunction, depending on the needs of the user.

Regardless of the purposes of the tools and analyses, they are all implemented in the form of plugins. A plugin may rely on another already existing plugin to perform its work, or it may solely use a core kernel *application programming interface* (or API) useful for, *e.g.*, parsing or for the exploration of *abstract syntax trees* (or ASTs). In the case of SIDAN, the analyses that compute our invariants have thus been implemented in the form of a plugin as well.

Obviously, not all of the Frama-C plugins are useful for our purposes; we quickly present the plugins we use in the next section, but apart from in this paragraph none of the others

will be mentioned. It however seems fair to say that Frama-C does not limit itself to the use we make of it, and that it is able to, *e.g.*, prove the validity of user-defined specifications of functions thanks to weakest precondition analyses, by the way of the *Jessie* and *Wp* plugins.

Finally, the whole framework is written in OCaml [Inr], and therefore so is our plugin for SIDAN.

6.1.2 Useful plugins for SIDAN

We give a quick description of the three Frama-C plugins on which the SIDAN plugin relies (although they all have been already mentioned in previous sections, at least implicitly) and give an insight of how they are useful for our purposes.

The value analysis plugin The value analysis plugin is a heavyweight plugin used by many others (including the two other plugins used by SIDAN, as a matter of fact). It is a powerful tool, albeit expensive, that allows the user to perform queries of the form "what are the expected values that *this one variable* may take at *this one precise point of the program*?". The answer comes in the form of a non-relational variation domain (that is a variation domain using only one variable at a time) which may be of several sorts, as discussed in section 5.3.

One should keep in mind that this type of analysis is not without limitations, not all of these being easy or even possible to overcome. In particular:

- The nature of this analysis needs the user to specify an entry point for the program. This is because the range of values that a variable (say *babar*) may take is in other things influenced by the range of values that may be taken by the arguments of the function in which *babar* is declared. The values of the arguments therefore need to have been determined in the first place, which again implies knowing the order in which the functions are called.

  It is of course possible to suppose that every function may be an entry point, but this would highly decrease the precision of the analysis; the alternative is then to specify an entry point. Most of the time this is not hard in itself —C programs generally use the *main* function as entry point— but it implies analysing the whole program all at once, which is not as convenient as would be a function-wise analysis.
- Not all of the data types are equally amenable to analysis by this plugin; although it performs well on integers, it is quite harder to efficiently track the value of pointers and, *e.g.*, to handle C strings. This is the reason why we mainly consider invariants on integers in this work.
- A sometimes pretty annoying limitation, which in a way is a consequence of the previous one, is that "degenerations" might occur when the analysis is unable to determine with enough precision where a given pointer points, and when this pointer is used to write in the memory. When this happens, from the point of view of the value analysis, the write instruction may overwrite any location in the memory and therefore nothing can be said about anything anymore.

If such a problem happens (and if it does indeed happen because of imprecisions in the analysis, not because there is actually a bug in the program), one should ideally run the analysis again on a program instrumented to help it being more precise.

- Frama-C is a framework for the analysis of C, and obviously not of assembly instructions. It is however common for the C compilers to permit the user to insert inline assembly code in the programs, making it impossible for the value analysis to deduce anything about variables modified by the assembly code.

  The two extreme ways for the value analysis to deal with such a situation are the following: considering that the assembly code may modify anything in the memory (thereby causing a degeneration as previously described); or considering that it has no effect whatsoever on any variable (this is the choice made by Frama-C). It should be obvious that none of these is satisfactory, making the analysis of programs containing inline assembly problematic.

  One solution to this problem is to painstakingly replace by hand every assembly code by its best (ideally equivalent, if any) C approximation with respect to the analysis.

- Finally, in the current state of Frama-C, no analysis of parallel code is possible.

Even in the presence of these limitations, the value analysis plugin is, as we already said, a very powerful tool. One of its main advantages is that it is logically sound; in other words, it means that the variation domains that it computes are always over-approximations of the *real* domains[20]. For instance, if the value analysis states that $var \in [0,127]$, it is guaranteed that we will never have $var = 224$. However, there is no guarantee on the precision of the analysis, and it may well be possible that *var* is in fact never different from 0.

At last, we should mention the fact that the precision of the value analysis is tunable by the way of a few parameters, one such being the *slevel* of the analysis. Without too much details, let us say that the value of the slevel determines how much the loops are unrolled *and* how far *if*/*then*/*else* branches should be separately analysed. A high slevel will make the analysis slower (if the code happens to contain big loops or intricate conditionals) but also most of the time more precise.

As the results of the value analysis are used in SIDAN (and the other plugins used by SIDAN), it goes without saying that the value of the slevel will greatly influence the efficiency and precision of the whole instrumentation. Unfortunately, there is no simple way to quickly determine the "right" slevel for a program.

The program slicing plugin It has already been said that we use program slicing in SIDAN as a way to collect variables of interest for running the value analysis. In practice, slicing is performed by a Frama-C plugin devoted to this task, appropriately called *slicing*.

There is not much issues when using this plugin in SIDAN as we do not care so much about what generally is important when computing a program slice, such as ensuring, *e.g.*,

[20] This is contingent on the fact that any problem related to the analysis has been resolved, *e.g.*, any inline assembly code has been properly dealt with.

that the resulting slice is a valid, compilable C program[21]. We also do not really use any of the options featured by the slicing plugin as they are mainly concerned with specifying "where" the slice is computed, and the location of the slices we are interested in can easily be specified from inside our own plugin.

It may be interesting to notice at this point that the fact that we use a whole program slice to gather variables does not —as one could suspect— actually lead to problems about the scope of the variables when they are eventually used in assertions. This is simply because the only variables from the slice that will have a valid variation domain (as computed by the value analysis) at the point where the assertion is inserted are the one that are always defined at this point of the program.

The data scope plugin The (data) scope plugin is used in a rather specialized setting inside SIDAN, as it is only useful for the computation of one sort of invariant (namely the one on locally constant variables, presented in section 5.5), whereas the two plugin previously presented are useful for all of what SIDAN does.

Among the few services proposed by the scope plugin, the one we used is its computation of so-called data scopes around variables. A data scope for a given variable V at a given point P of the program is defined as the set of statements in the program where V holds the same value as at point P; it should be obvious that this sort of sets may be very useful when computing the invariants of section 5.5.

As a matter of fact the scope plugin actually defines two slightly different sorts of data scopes: the backward scope is made of the statements S_i for which the value of V is identical between the execution of the S_i and the next execution of P; the forward scope is made of the statements S_i for which the value of V is identical between the execution of P and the next execution of the S_i. This is illustrated in figure 12, where statement 2 is the only one in the forward scope for the variable var_1 at statement 3; and where statement 3 is the only one in the backward scope for the variable var_1 at statement 2.

$$1 : var_1 = 0; \quad 2 : var_2 = 1; \quad 3 : var_1 = var_1 + var_2;$$

Figure 12: A tiny program to illustrate data scopes

In the case of our invariant, what we seek are variables that have the same value after the execution of a statement consisting in a function call. One way to put it in terms of backward and forward data scopes is to say that for a variable to be part of an invariant, we need the function call to be part of the backward data scope computed for this variable at the statement immediately following the function call.

[21] This is in fact something that is guaranteed by the plugin, thus a serious user of the plugin should not have to worry about this either. The point is, from our point of view we do not even care whether this property holds or not.

6.1.3 Frama-C and the C Intermediate Language

We now turn to an important component of the Frama-C kernel that is also central to our plugin, that is the *C Intermediate Language*, or "CIL" for short[22] [NMRW02]. CIL is a high-level intermediate representation of C programs aimed at easing source code to source code transformation and program analysis. Like Frama-C, CIL is written in OCaml.

Representing C programs in CIL is useful for a range of reasons; the most down-to-earth are probably that CIL uses a simpler, less-ambiguous syntax than C, and that it defines a clean API to handle C instructions and types in OCaml. Another interesting feature of CIL is that it features a series of "tree visitors" useful to explore (and potentially transform) the abstract syntax tree of a program. We detail a few of these points in the next paragraphs.

Unsugar the code CIL performs various "simplifications" of C programs. In other things, it transforms all the sorts of loops in a single loop construct, and it replaces syntactic sugar constructions such as "`->`" by their alternatives.

These transformations are important to keep in mind in the context of Frama-C, because even if they do not change the semantics of the code, they may introduce various temporary variables that are not present in the original program. The code subsequently analysed by Frama-C being the CIL version, one should be prepared to interpret the result of the analysis with respect to the original program.

In the case of SIDAN, it means that we should either be able to remove temporary variables introduced by CIL from the assertions we will insert in the code, or to accommodate the instrumented program so that the assertions make sense. It is the second solution that we chose in our case, as it will be made clear later.

An insight into the abstract manipulation of C instructions CIL features an OCaml API that permits the user to conveniently reason both on the nature of C instructions and on the type of the involved variables.

For instance, one important CIL object is its representation of C *left values* (or "lvalues" for short); those are represented by an *lhost* (which is most of the time a variable) and an *offset* which —when not null— represents the index of an array or the field of a structure.

There is also CIL objects for higher level elements like instructions or statements. An instruction, for instance, may be the assignment of a an lvalue from the value of an expression (yet another CIL object), and a statement may embed an instruction or, *e.g.*, conditional or loop constructions.

Finally, C types themselves are CIL objects, including function types and structures. It is therefore quite easy to perform analyses conditional to the types of variables. Properly understanding CIL objects is important to implement SIDAN, as they are precisely the objects on which the analyses of Frama-C are eventually computed.

CIL AST visitors The CIL visitors of the abstract syntax tree are in general not directly used inside Frama-C, as the latter defines a series of customized visitors that overload the ones

[22] Not to be mistaken with .NET's *Common Intermediate Language*.

from CIL. Nonetheless, the nature of these visitors does not change much.

We will not go too much through the details of the visitors, but we think it is worth mentioning that the user may choose several levels of granularity when visiting an AST. He may indeed choose to go through the code of a program statement by statement, or instead at the level of instructions. This choice of course depends on what the user tries to achieve, and even then, there might not be a choice strikingly better than any other.

Another thing that should be said about visitors is that they may be used both to perform on-the-fly transformations of the AST (and hence of the source code of the program), or to perform a side-effect free visit. We will not discuss the details of the different possible behaviours for a visitor, though.

6.1.4 Value analysis internals, a quick overview

In the same way as we described some elements of the C Intermediate Language in the previous section, we now present a few things about the internal use of the value analysis plugin in Frama-C. Although quite low-level, these points are important in practice in order to be able to make good use of Frama-C. Understanding these took an important part of the implementation effort of this work.

The process of using the value analysis of Frama-C can be quite faithfully described in the following way:

1. one should start by considering the variable and location in the code for which he wishes to obtain results from the value analysis; this has mostly to do with the CIL representation of the program as explained in the previous section;
2. one should then call upon the value analysis to actually obtain something useful for his purposes;
3. finally, one should take profit of the results of the analysis in whichever way he wants.

The second point is actually a little bit trickier than what might first appear; one of the original objectives of this work was precisely to improve how the value analysis was used so that arrays and structures could also be analysed like integers. However, it is also relevant to see this stage in a high-level fashion as the conversion from a CIL *Lval* to a Frama-C *Ival* —the custom OCaml type that aggregates the different possible types of results of the value analysis.

It is also at this point that one may customize the value analysis in order to obtain the invariants on aggregated variables presented in section 5.1.

The third point has to do with the proper use of the *Ival* results. Those are in fact representations of the variation domains of variables, as they were described back in section 5.3. The two simple forms an Ival may take at this point when used in SIDAN are precisely a *Set* of integers and a *Top*[23], which really is the name of the interval results.

An Ival may actually also represent results on floating point types, but those are violently ignored in SIDAN, in part because performing comparisons on float C types is tricky.

[23] The name “Top” comes from the $\top$ top symbol usually used in abstract interpretation to denote the greatest element of the lattices.

6.1.5 ANSI/ISO C Specification Language "ACSL"

This section presents yet another aspect of Frama-C, that is the *ANSI/ISO C Specification Language* or ACSL [CEAa]. Although not really central to our argument, ACSL is an ubiquitous view in Frama-C, especially when one is involved with the value analysis. It is a versatile specification language that allows a user to express necessary conditions on variables at some point of a program, or highly precise specification of functions, among other things. Both of these aspects may be visible in SIDAN, although we delay the discussion of the latter to section 6.3.2.

It is not our objective to give a detailed presentation of ACSL annotations; it is however a good pretext to give us a little further insight into the value analysis.

Form of the ACSL annotations The ACSL annotations may take many forms; a commonly encountered one is ACSL "assertions", quite naturally introduced by the *assert* keyword. A wide range of properties may be asserted on variables, such as the fact that they are initialized, that a pointer points to valid memory, or simply a numerical property like "the value taken by this variable is an odd integer".

Some forms of annotations may be more specific to function specifications, such as the *requires* and *ensures* keywords which permit to define typical function contracts where some property is guaranteed after the execution of the function, given that some properties held before it was executed.

An example of a simple ACSL assertion is shown in figure 13, and states the fact that two memory locations are valid before being dereferenced. One may note that ACSL annotations are inserted in the form of comments —like many others, such as the Java annotations.

```
/*@
        assert \valid st->a;
        assert \valid st->b;
*/
st-> a = st->b;
```

Figure 13: A simple ACSL assertion

ACSL and the value analysis The reason why ACSL annotations (and more specifically assertions) are so much present when using Frama-C's value analysis is that the soundness of the results of the analysis is most of the time conditional to the validity of some properties of the program variables (we already quickly mentioned a *caveat* about the soundness of the value analysis in section 6.1.2).

This means that the value analysis will routinely generate assertions of the form given in figure 13, and it is the responsibility of the user to ensure that those are actually valid (he may "convince himself" that they are true, but the cleanest way to deal with such assertions is of course to prove them, *e.g.*, with the help of other Frama-C plugin such as Jessie or Wp).

As we do no such thing in SIDAN, we therefore do not truly guarantee the soundness of the assertions that we create, and there is always a risk for an assertion to actually be false.

6.2 Structure of the plugin

This section now gives a high-to-mid-level view of the implementation of our SIDAN plugin. We start with two examples of instrumentations that show graphically when Frama-C plugins are used and to what purpose.

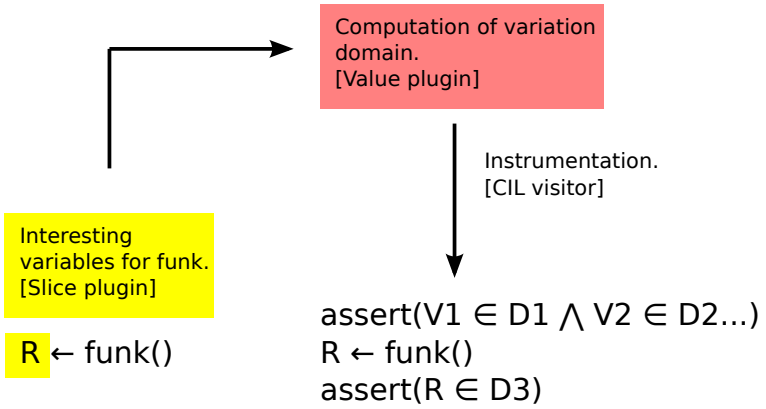

Figure 14: Process of a SIDAN instrumentation (type 1)

The instrumentation from figure 14 uses invariants presented in section 5.1 and in section 5.4. Figure 15 uses the invariants from section 5.5.

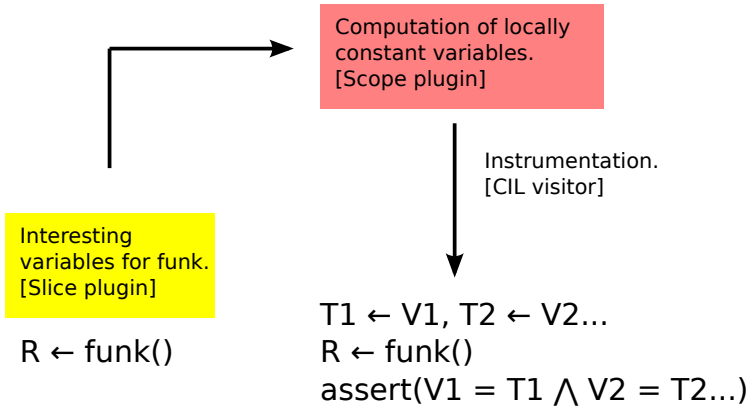

Figure 15: Process of a SIDAN instrumentation (type 2)

We also show a lower-level partial function call graph in figure 16 that highlights the important parts of our plugin, function-wise.

We give a quick description of these functions presented in figure 16; their purposes and structure should be quite clear from what has already been said in section 6.1.

vstmt_aux The function *vstmt_aux* is a CIL visitor at the statement level which is at the core of the instrumentation. It finds the function calls which we wish to instrument and delegates the invariant computation to the relevant functions. It is also the function responsible for the actual modification of the AST by replacing the original function call statement by an instrumented one.

get_slice The function *get_slice* is more or less an API call to the "PDG" program slice plugin to compute a specific program slice, tuned to our needs in the case of SIDAN.

create_exp The function *create_exp* is a major component of SIDAN and is tasked with building a CIL expression that represents the about-to-be-inserted invariant for a given left value at a given point of a program.

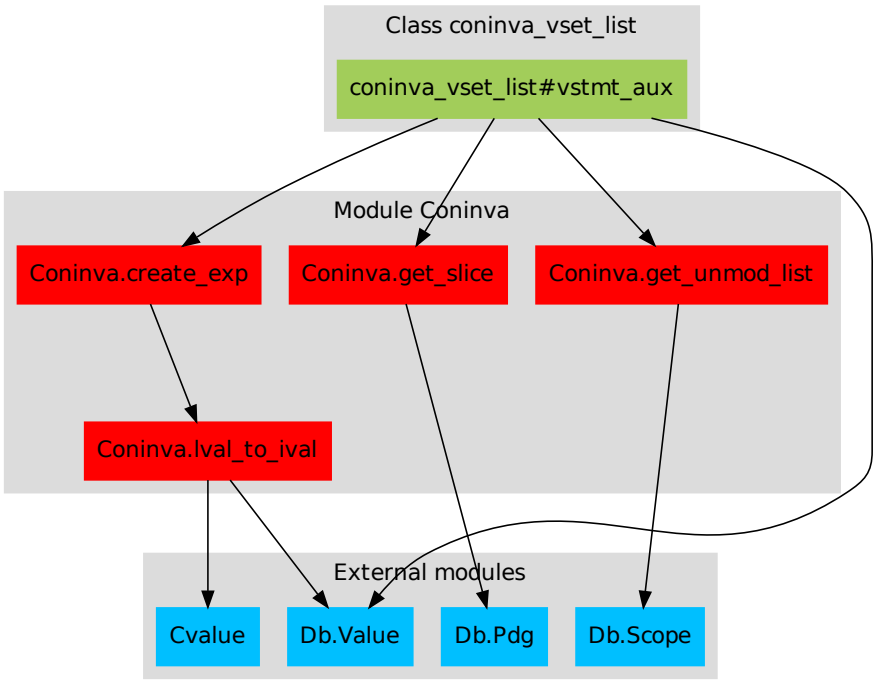

Figure 16: Call graph of the main functions of SIDAN's implementation

lval_to_ival The function *lval_to_ival* does the conversion work needed to obtain a variation domain (as computed by Frama-C's value analysis) for a given left value. This is why it relies on Frama-C's value plugin and on its internal representation of left values and analysis results.

get_unmod_list The function *get_unmod_list* implements the invariant on locally constant variables from section 5.5 by internally relying on Frama-C's scope plugin.

We do not think it would make much sense at this point to describe SIDAN's structure and implementation in more details; we recall however that the SIDAN plugin is an open-source software, and as such its source code is[24] made freely available on the Internet on `http://www.rennes.supelec.fr/ren/rd/cidre/tools/sidan`.

6.3 Implementation details

This section is concerned with what is maybe the darkest side of this work, that is the issues that were faced (and had to be solved) entirely in the context of SIDAN's implementation. We think however that the time spent working on them justifies plainly their being mentioned in this report.

6.3.1 Shifting right in the periodic table

The very first implementation issue was to update the original SIDAN plugin from [Dem11] so that it would work with the newest version of Frama-C. When we started our work, there was a SIDAN plugin for the *Beryllium* version of Frama-C and one for *Carbon*, its successor.

The newest version of Frama-C, *Nitrogen*, was released in October 2011. It features impressive improvements from Carbon, notably the fix of an important memory leak that unnecessarily hampered the value analysis. Considering the extent of the improvements, it was the most logical thing to do to port the original plugin to Frama-C Nitrogen.

The extent of the modifications we had to make on the SIDAN plugin for it to work under nitrogen was not so large[25], although the relative lack of documentation both on the original plugin and on Frama-C itself made it sometimes difficult to determine how the code had to be modified. The biggest issues were mostly on identifying the parts of our plugin that —even if functionally correct— could benefit from updates in the Frama-C API to be implemented again in a more efficient fashion.

We also sometimes had to face what were apparently regressions from former versions of Frama-C. For instance, we faced an unexpected bug where the translation of original C code to CIL performed by Frama-C would sometimes introduce temporary variables without declaring them beforehand. This bug was thankfully quickly fixed by the Frama-C team and did not cause us lasting problems, though.

[24] Or rather *will be*, as soon as possible.

[25] These modifications were mostly matters of changing some types and updating some API calls to reflect the new organization of some modules inside Frama-C.

6.3.2 Stubs intermission

We already mentioned several times the fact that the value analysis of Frama-C is no out-of-the-box miracle, in the sense that one may need to prove assertions or simulate the behaviour of non-analysed code for it to be sound.

An important part of the work that needs to be done to accommodate the Frama-C ways is the creation of function stubs. These stubs are quite thoroughly described in the original work on SIDAN [Dem11] and it was not the main goal of our work to modify them so much. We did however slightly revisited the concept and tried to experiment and see if "cleaner" function specifications *à la* ACSL could be substituted to the stubs.

We first briefly recall the motivation for stubs and their original structure, and then show on an example how ACSL specifications can help solving concrete problems that may arise during the value analysis.

Stub your way The purpose of the stubs is to properly fake the behaviour of library function that may be used by programs being instrumented by SIDAN. Without stubs, the default behaviour of Frama-C when faced with external functions is more or less to ignore them and to consider that they meet some general properties.

This is most of the time unsatisfactory, because one assumption made by Frama-C is that external functions are side-effect free, *i.e.*, they never change the value of the memory pointed to by their arguments, if any. This is obviously not always true; one way to remedy this problem is to pretend that the problematic functions are not external, but instead defined by their stubs alternative.

Stubs might also be relevant to simulate the execution of inline assembly code, as it was already mentioned in section 6.1.2.

The implementation methodology used for stubs in [Dem11] is straightforward, and consists in defining alternative functions as simple as possible that act like the original in term of return values and side effects. They also take into consideration the error values that the function might return, and in particular the *errno* values that it might set.

We recall that errno is a global integer variable used in C programs to give information about the nature of an error; C functions typically do not include such information in their return values, and it is therefore necessary to read the value of errno to determine if, *e.g.*, a call to *read* failed because its file descriptor argument is not valid or because the call was interrupted by a signal, among other things.

Well-written software tend to handle errors properly, and therefore rely a lot on the different values errno might take; hence it is important for the stubs to faithfully emulate this behaviour as well.

This approach to stubs is fairly comprehensive but it is not without its own drawbacks.

First, as the stubs are designed to fool Frama-C, it implies that they need themselves to be analysed by the value plugin for them to be effective. Even if the stub functions are in general quite small, the value analysis is still an expensive process and thus using (many) stubs leads to additional cost when performing the analysis on the whole program.

Second, a change in the way assertions are inserted (we discuss this issue in more details just in the next section) implies that the stub functions need now to be cleaned up once the instrumentation process is finished. This is far from being something difficult to do manually, but automating the process is probably a small challenge of its own (we have not tried yet, though) and it is in any way something annoying to do. This issue will be made more explicit in next section.

An ACSL example We now give a quick example of how ACSL specifications can be used instead of stubs as far as we are only concerned with their return values and side-effects. We did not investigate the issue so much, and therefore did not try yet to use ACSL to express information about errno values. This is why those specifications cannot entirely replace the existing stubs for the moment.

We use the example of the *read* system call, for which the complete prototype is: `ssize_t read(int fd, void *buf, size_t count);`. It is obvious that when successful, *read* will write in the memory zone pointed to by *buf*; yet the default behaviour of Frama-C is to ignore the fact that up to *count* bytes of *buf* may be modified. A simple ACSL specification that partially solves this problem is shown in figure 17.

```
/*@
        requires \valid((char *)buf);
        assigns *(char *)buf;
        ensures ((\result <= count && \result >= 0) ||
                 (\result == -1));
*/
extern ssize_t read(int fd, void *buf, size_t count);
```

Figure 17: ACSL specification of the behaviour of the *read* system call

We can note that the specification of figure 17 is less precise than the actual possible behaviour of *read* (and than what could be inferred from the analysis of the *read* stub), as it omits information on "how much" *buf* is modified (*i.e.*, from zero to *count* bytes); it also does not give any insight about errno, as it was already said. However, it still provides a few useful information about, *e.g.*, the possible range of result values for the function.

In the end, ACSL specifications are quite cleaner than stubs, in the sense that they do not really impact the performance of the analysis as the stubs did, and that they are more discrete when inserted in the code. It would thus be a logical objective to eventually replace all the stubs with such constructs.

6.3.3 Inserting assertions in the abstract syntax tree

We conclude our discussion of the implementation work with this section, dealing with how the instrumentation of programs is eventually performed.

We start by recalling that the original SIDAN plugin of [Dem11] performs the instrumentation in two steps by the way of the *patch* utility program. What it does is creating "diff" files (one for each source file) containing all the assertions that have been created along with the location in the source code where the assertion needs to be inserted. The latter information can easily be obtained thanks to the CIL component of Frama-C. Then, it simply patches each original source file with the corresponding diff, which results in an instrumented source code, ready to be compiled.

This approach works quite well in practice, and it results in instrumented code that can quite easily be compared with the original. However, it relies on a pretty low-level information to work, *i.e.*, the location (basically a line number) in the code of any function to be instrumented.

This led us to a problem we did not expect when we started experimenting with invariants that wanted assertions inserted *after* the function calls. Indeed, the naive approach of inserting the assertion on the line after the one of the call is foiled when this one is written on several lines (which happens quite often in practice); writing the assertion on the line before the statement immediately following the call does not work either, not least because there is no guarantee that a function call is always followed by a statement (and in particular a statement belonging to the same scope).

Instead of trying to determine a reliable way to derive the line number where "post-assertions" can be inserted from a function call, we decided that it could be better to rethink the whole way of how assertions were inserted. It seemed sensible to make a deeper use of the CIL tools offered by Frama-C and to directly insert the assertions by modifying the program AST. As it was already said, CIL allows to do this quite efficiently by the way of an AST visitor; and changing our plugin to generate assertions in the form of CIL expressions (to be inserted in the AST) instead of strings to be printed in a file was expected to be quite straightforward (and was indeed).

The advantages of switching from patching to AST modification come mostly from the fact that the control of the code transformation becomes semantic instead of being purely syntactic. For instance, inserting assertions after function calls becomes pretty straightforward, and it is not really more complicated than doing it before the call.

However, this approach is not without disadvantages of its own. First, it becomes trickier to match the code resulting from the instrumentation with the original one, because the instrumentation is now done on the code that has already been transformed by CIL (with everything that entails: single type of loops, additional variables, less sugar...). Also, it should be noted that CIL produces a single modified C file for *the whole program*, which makes function matching even more annoying. Second, because of CIL producing a single file, it is in general not possible to directly compile the source resulting from the instrumentation anymore, as it was the case with patches. This issue comes from the use of stub functions that was discussed in section 6.3.2: as there is no way for CIL to guess that those functions are not to be used *in real life* they are included in the transformed program as well, and therefore need to be removed before the compilation.

Therefore, we could say that inserting assertions by AST transformation is more flexible from the point of view of the plugin, but it is more costly to deal with from the point

of view of a user wishing to actually instrument programs. However, there may be ways to accommodate part of this additional cost like, *e.g.*, replacing all the stubs by ACSL specifications (in this case, only the annotated function prototypes would make their way to the transformed code, and they do not need to be cleaned up before compiling the program).

7 Evaluation

We shall now turn our attention to how the instrumentation produced by SIDAN is actually useful in the *real life*. In particular, we would like to see how the various improvements we made on the invariants impact the global result.

This evaluation should be carried on three criteria: how costly it is to instrument a program; how efficient is the generated instrumentation; and how much overhead is added by the instrumentation?

Unfortunately, we did not have as much time as we would have liked for the evaluation presented in this work to be thorough enough; in particular, what we did was quite less comprehensive than what is done in the original work on SIDAN, which makes the comparison between the two a bit tricky.

The main omission during the evaluation of our work from what was done in [Dem11] is the experimental test of the assertions by fault injection, which allowed to obtain rather precise figures on how sensitive to attacks were the assertions.

7.1 Performance of the instrumentation process

The evaluation of the performance of the instrumentation process is not easy, because there is typically a huge variability in the time needed to perform the value analysis from one program to another (even when comparing programs of the same number of lines). For instance, figures given in [DMTT11] show that a program of 2300 lines was fifteen times longer to analyse that one of nearly 3000 lines.

In our case, there were no reasons to believe that the instrumentation process would take a time significantly different from what would take the original implementation of SIDAN. We did not try confirming this hypothesis in a rigours fashion, however, in part because it would have needed us performing instrumentations from the original and new implementation in an identical and controlled environment. We can still informally state that this hypothesis seems to be valid, from what we witnessed in our experiments.

In any case, the impact on the performance of the modifications we brought to SIDAN are dwarfed by the improvements coming from the updates of Frama-C, in which we are not really involved. For instance, the memory consumption of the value analysis plugin decreased from several giga-bytes down to a few hundred mega-bytes when switching from Frama-C Carbon to Frama-C Nitrogen.

7.2 How do the new invariants fare?

Unlike in [Dem11], we did not directly evaluate the impact of our invariants. We try to circumvent this issue by directly comparing what invariants we generate with the ones that were created by the original SIDAN, when used on an identical program.

This permits us to both compare the potential increase in preciseness for invariants existing in the two instrumentations and to witness where new invariants might have been created. Unfortunately, it involves some guesswork if one wants to know the impact of those differences.

We chose to compare our invariants by instrumenting the *ihttpd* web server [Jør]. The choice for ihttpd (from among the software for which there is an existing instrumentation by SIDAN to which we could compare ours) was motivated by the fact that it is a quite short program (hence making it quicker to analyse) and that it seems quite relevant to harden a web server against attacks, as it is a typically well-exposed sort of software.

The analysis of ihttpd in [DMTT11] reports 145 assertions generated by SIDAN. In the case of our instrumentation, this more than doubles to 316, which is a nice improvement even if we have to keep in mind the fact that a lot of those new assertions are inserted after the call of already-instrumented functions (and therefore, we do not double the number of instrumented functions).

We now give a series of examples of invariants that highlight our improvements on SIDAN. In figure 18, we compare the original assertion (shown on the left) and the new ones for a function call of ihttpd. The new assertions show the use of intervals in the invariants and feature an invariant on the return value of a function as well.

```
coninva_assert(argc == 2);

t2 = strcmp(*(argv + 1), "--vers");
```

```
coninva_assert(t1 >= -117 && t1 <= 210
                && argc == 2);
t2 = strcmp(*(argv + 1), "--vers");
coninva_assert(t2 >= -45 && t2 <= 210);
```

Figure 18: Original (left) and new (right) assertions for a function call in ihttpd

Figure 19 shows an example of invariant on an array that did not exist in the original instrumentation. In this example, the variable *access_log_filename* is an array of *char* (*i.e.*, a C string), and the invariant therefore simply states that *access_log_filename* is the empty string.

```
coninva_assert(access_log_filename[0] == 0);
access_log = fopen("/dev/null", "w");
```

Figure 19: An assertion using an array

Figure 20 shows an example of invariant partly resting on a structured variable. It shows how this can be used both for an original invariant using a variation domain and for an invariant on locally constant variables.

```
{
        register in_addr_t __babar171;
        __babar171 = sa.sin_addr.s_addr;
        coninva_assert(sa.sin_family == 2 && t1 == 0);
        ip = ntohl(sa.sin_addr.s_addr);
        coninva_assert(sa.sin_addr.s_addr == __babar171);
}
```

Figure 20: Assertions on variation domains and locally constant variables

7.3 Instrumentation overhead

We end this part on the evaluation with a short discussion on the performance overhead on the program execution added by the assertions. We did not carry benchmarks as comprehensive as in [Dem11], which used the SPEC CPU2006 benchmark [Cor06]. This evaluation found a quite low overhead, of the order of a half percent to one percent decrease in a SPEC score[26].

The results of this evaluation cannot be used directly in our case, because we may add quite a few invariants that may make the whole assertion checking become costlier. However, we could argue that the total overhead due to these new invariants should not be tremendous, as even doubling the original overhead would still be quite reasonable.

However that may be, we feel that it is still nice to provide some quick benchmarking of our own. To this purpose, we used an instrumentation of a reference (non-optimized) implementation of the SIMD hash function [LBF09] in its 256-bits setting, that already features some simple performance testing.

Our instrumented version of SIMD adds 51 assertions to a source code of the order of roughly 1000 lines. When ran, the program computes an estimation of the throughput of the hash function and an approximate number of cycles needed to hash one byte of data. We show the performance figures for the raw and instrumented program in table 1.

Version	**Throughput** (Mbytes/s)	**Efficiency** (cycles/bytes)
Raw SIMD256	1.006	2478
Instrumented SIMD256	0.995	2506

Table 1: Comparative speeds of SIMD256 with and without instrumentation

The results from table 1 tend to confirm the idea that the overhead of SIDAN's instrumentation is very low; the instrumented version of SIMD achieves a throughput equal to

[26] A SPEC score measures the ratio between a base running time for a test program and its actual running time when being tested. A difference of one percent between two SPEC scores is therefore pretty negligible.

0.989 of the original, and the amortized additional cost of the instrumentation is only 28 cycles per byte of data.

When evaluating the overhead, it should also be noted that given the modest impact of SIDAN's assertions on the execution time of a program, too thorough an analysis of its direct overhead would not make so much sense. This is because a *real life* use of SIDAN would likely imply a more important indirect overhead coming from the safeguards that would ensure that SIDAN's assertions could not be bypassed. This safeguarding can easily be done thanks to control-flow monitoring methods, but those would add an overhead of their own that is typically more (or at least equal) than SIDAN's (as, *e.g.*, in [KBA02, ABEL05a]). Therefore, only benchmarking the combined overhead would really be useful for an end-user.

8 Tutorial essentials

We conclude this report by a quick tutorial that explains how to use our plugin to actually instrument a program. The instrumentation process itself is partly automatic, but it requires some familiarity with Frama-C and SIDAN options.

8.1 Basic Frama-C options

Slevel We already discussed the *slevel* option in section 6.1. It is a quite important tuning parameter but it may be hard when first instrumenting a program to know to which value it should be set; choosing the slevel hence often involves guesswork.

Calldeps The *calldeps* option is quite important too, and should always be set. It ensures that the functional dependencies computation of the value analysis will never try to simplify and merge the dependencies of a function for multiple calling sites.

This is important because we instrument each function call separately, and their dependencies provide us with the variables on which computing invariants.

Val-ignore-recursive-calls This option ensures that Frama-C will not stop its analysis when encountering a recursive function call. The current value analysis of Frama-C does not deal with such calls, and the default behaviour when encountering one is simply to stop the analysis, as there would be no guarantees on the soundness of the result if it were to continue.

In our case, a bit of unsoundness is still acceptable, and it is therefore better to continue the analysis than producing no result.

More options! Always more options! There is of course a range of Frama-C options different from the few we mentioned here; they might not be immediately useful when first using SIDAN, though, and any motivated user should really read Frama-C's documentation at some point to determine by himself what options are relevant in his case.

8.2 SIDAN options

We describe here the few SIDAN options that may be used to tune a little bit the instrumentation that is performed. SIDAN itself is implemented as a plugin in Frama-C; it is invoked with the option *coninva*.

Tuplevalue The *tuplevalue* option is the one that triggers the use of invariants on aggregated variables discussed in section 5.1.

Retart This option enables the creation of invariants using the return values of functions, presented in section 5.4.

Postart This option controls the invariants on locally constant variables of section 5.5.

Noinj Finally, this option prevents the insertion of the code used for fault injection, which is only useful when one wishes to perform some tests on the assertions produced by SIDAN on the program being instrumented.

8.3 Selecting files and printing the result

It was already mentioned that the value analysis of Frama-C is best performed from a relevant entry point, typically the *main* function in C programs. A corollary is that the analysis (and hence SIDAN) needs to be ran on all the files part of the instrumented program at once.

In practice, this can be achieved quite efficiently for programs using a *Makefile*. The original SIDAN tools came with a script that uses the *make -n* "pretend" option to list all the files part of the program and give them as an input to Frama-C. To these files, one also needs to add the stub functions and ACSL specifications described in section 6.3.2.

In order for the result (*i.e.*, the modified AST of the program, represented by its C syntax) to be printed, one simply needs to use Frama-C's *print* option, probably along with the *ocode* option as well, that permits one to specify an output file.

8.4 The Final Touch

Once the file resulting from the instrumentation has been output, it is still necessary to do a little bit of post-processing before being able to compile it. Part of this is easy, like adding the declarations of the functions one wishes to use to check the assertions; but other parts are more annoying, like removing the code of the stub functions (the reason for this being presently necessary was discussed in section 6.3.3).

8.5 Command-line example

We show a simple example of a SIDAN command-line invocation to instrument a program made of a single file in figure 21. This example omits the inclusion of the additional stubs and ACSL files, but includes most of the options discussed in this section.

```
frama-c -verbose 0 -calldeps -slevel 299 -coninva \
        -retart -postart -noinj katan.c \
        -print -ocode katan.sid.c
```

Figure 21: Command-line invocation of Frama-C on a single file

9 Future work

An immediate future work to improve on what has been presented in this report would consist in thoroughly evaluating the impacts of our modifications to SIDAN. We did perform some evaluations, that were presented in section 7, but time constraints prevented us to lead what would have been interesting experiments. For instance, it would be much relevant to realize a fault injection campaign similar to what was done on the original SIDAN.

Another future work would be further investigating the use of ACSL for the definition of stub functions, as this was outlined in section 6.3.2. The use of ACSL specifications for library functions could be a clean alternative to the current stubs.

It would also make sense to try finding reasonable solutions to the problems that arose due to the instrumentation by direct manipulation of the programs' ASTs. These issues were mentioned in section 6.3.3, and we did not think of any clear way to solve them for the moment.

Finally, there is probably still room for new, as yet unthought-of invariants that could be implemented thanks to Frama-C. Searching for them would certainly be part of any future work on SIDAN.

10 Conclusion

This report described our work on SIDAN. We presented improved and new invariants for intrusion detection: we extended the original invariants of SIDAN so that they could involve arrays, structured variables, and variation domains represented as intervals. We then introduced new invariants based on the return value of functions and on the variables that are left unchanged by the function calls. We also described the implementation part of our work which mainly consisted in implementing the improved and new invariants in the SIDAN Frama-C plugin.

References

[ABEL05a] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. In *12th ACM Conference on Computer and Communication Security*, CCS '05, New York, NY, USA, November 2005. ACM.

[ABEL05b] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control flow. In *Formal Methods and Software Engineering*, volume 3785 of *Lecture Notes in Computer Science*, pages 111–124. Springer Berlin / Heidelberg, 2005.

[AP02] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

[BCS06] Sandeep Bhatkar, Abhishek Chaturvedi, and R. Sekar. Dataflow Anomaly Detection. In *Security and Privacy, 2006 IEEE Symposium on*, may 2006.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[CCH06] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association.

[CEAa] CEA. ANSI/ISO C Specification Language. http://frama-c.com/acsl.html.

[CEAb] CEA. Frama-C. http://frama-c.com/.

[Cor06] Standard Performance Evaluation Corporation. CPU 2006, 2006. http://www.spec.org/cpu2006.

[CXS+05] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, Security '05, Berkeley, CA, USA, 2005. USENIX Association.

[Dem11] Jonathan-Christopher Demay. *Génération et évaluation de méchanismes de détection des intrusions au niveau applicatif*. PhD thesis, Université de Rennes 1, 2011.

[DMTT11] Jonathan-Christofer Demay, Frédéric Majorczyk, Eric Totel, and Frédéric Tronel. Detecting illegal system calls using a data-oriented detection model. In *Future Challenges in Security and Privacy for Academia and Industry*, volume 354 of *IFIP Advances in Information and Communication Technology*, pages 305–316. Springer Boston, 2011.

[GRSRV03] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante. Soft-error Detection Using Control Flow Assertions. In *Defect and Fault Tolerance in VLSI*

Systems, 2003. Proceedings. 18th IEEE International Symposium on, DFT '03, pages 581 – 588, nov. 2003.

[Inr] Inria. The Ocaml programming language. http://caml.inria.fr/ocaml/.

[Jør] Ivan Skytte. Jørgenssen. The ihttpd HTTP server. http://i1.dk/download/ihttpd.

[KBA02] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the 11th USENIX security symposium*, Security '02, pages 191–206, 2002.

[LBF09] Gaëtan Leurent, Charles Bouillaget, and Pierre-Alain Fouque. SIMD Is a Message Digest. SHA-3 submission document, 2009.

[LLHT11] Romaric Ludinard, Loïc Le Hennaff, and Eric Totel. RRABIDS, un système de détection d'intrusion pour les applications Ruby on Rails. SSTIC '11, pages 266–278, 2011.

[Min04] Antoine Miné. *Domaines numériques abstraits faiblement relationnels*. PhD thesis, École Polytechnique, 2004.

[NMRW02] George Necula, Scott McPeak, Shree Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 209–265. Springer Berlin / Heidelberg, 2002.

[PAG] MIT Program Analysis Group. The Daikon invariant detector. http://groups.csail.mit.edu/pag/daikon.

[VA06] Ramtilak Vemu and Jacob A. Abraham. Ceda: Control-flow Error Detection through Assertions. In *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*, IOLT '06, 2006.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.