Slicing for Security of Code

Benjamin Monate and Julien Signoles[⋆]

CEA LIST, Software Reliability Labs,
91191 Gif-sur-Yvette Cedex, France
`Benjamin.Monate@cea.fr`, `Julien.Signoles@cea.fr`

Abstract. Bugs in programs implementing security features can be catastrophic: for example they may be exploited by malign users to gain access to sensitive data. These exploits break the confidentiality of information. All security analyses assume that softwares implementing security features correctly implement the security policy, *i.e.* are security bug-free. This assumption is almost always wrong and IT security administrators consider that any software that has no security patches on a regular basis should be replaced as soon as possible. As programs implementing security features are usually large, manual auditing is very error prone and testing techniques are very expensive. This article proposes to reduce the code that has to be audited by applying a program reduction technique called *slicing*. Slicing transforms a source code into an equivalent one according to a set of criteria. We show that existing slicing criteria do *not* preserve the confidentiality of information. We introduce a new automatic and correct source-to-source method properly preserving the confidentiality of information *i.e.* confidentiality is guaranteed to be exactly the same in the original program and in the sliced program.

1 Introduction

Bugs in programs implementing security features can be catastrophic: they may be exploited by malign users to gain access to sensitive data for example. These exploits break the confidentiality of information. All security analyses assume that softwares implementing security features correctly implement the security policy, *i.e.* are security bug-free. This assumption is almost always wrong and IT security administrators consider that any software that has no security patches on a regular basis should be replaced as soon as possible. As programs implementing security features are usually large, manual auditing is very error prone and testing techniques are very expensive. This article proposes to reduce the code that has to be audited by applying a program reduction technique called *slicing*. Slicing transforms a source code into an equivalent one according a set of criteria (see [13, 14, 12, 16] for surveys about slicing). We show that existing slicing criteria do *not* preserve the confidentiality of information. We introduce a new automatic and correct source-to-source method properly preserving the

[⋆] This work has been supported by the french RNTL project CAT ANR05RNTL00301 and the european ITEA2 €-Confidential project ITEA05011/N062930329.

confidentiality of information *i.e.* confidentiality is guaranteed to be exactly the same in the original program and in the sliced program. In the following sections this method will be called *confidentiality slicing criterion*. Thanks to this criterion, very drastic reductions can occur because usually only small parts of a program have an impact on security.

The advantages of these reductions are twofold. Firstly, they ease all auditing activities and help to focus on pertinent parts of the considered program. Secondly, whenever the confidentiality slicing criterion is not efficient (that is whenever the code is barely reduced), it points out that security features are scattered everywhere in the program and therefore are very error prone [1]. Moreover if a well-identified security-relevant part of the program is sliced, then it certainly contains major security issues because all security related parts of the program are kept unsliced.

For these purposes, we focus on an automatic over-approximated-but-correct source-to-source slicer. So, regarding the confidentiality criterion, it automatically transforms a compilable program p into another one p' which:

– is equivalent to p (in particular p' is compilable and has the exact same level of confidentiality ensured); and
– may contain useless code from a confidentiality point of view. This is the usual price for automation and correctness for undecidable problems.

This article focuses on confidentiality and integrity in source codes but deliberately ignores programming language level security issues like invalid pointer and array accesses also known as *buffer overflows*. One of our goals is indeed to formally study all security properties of C code as described by G. Heiser [6], but buffer overflows can be checked by standard *safety* verification techniques [3, 8].

Outline Firstly, we give a running and motivating C code example and describe its expected security properties. Secondly, we explain why the standard slicing methods cannot deal with these properties, especially confidentiality. Thirdly, we propose a new slicing technique to perform code reduction according to these properties.

2 Running example

The C code in Figure 1 is used as a running example in the following sections. It contains the definition of function `check_and_send` which sends a message `msg` under some specific conditions.

This code contains special comments between `/*@` and `*/`. The first one is above the prototype of function `send` and describes its expected security property: that is the sent `data` has to be `public` (*i.e.* readable by anyone if you are interested in confidentiality). The second one is above the prototype of function `publicise` and explains that, after a call to this function, its argument becomes `public`. The last special comment is a special cast `/*@ (public) */`: that is a

```
/*@ requires security_status(data) == public; */
void send(const void *data, const int dst[4]);

/*@ ensures security_status(data) == public; */
void publicise(char *data);

int check_and_send(char msg[]) {
  int src[4];
  int dst[4];
  int result = 0;

  compute_src(src);
  compute_dst(dst);

  if (dst[0] >= 190 && dst[0] <= 200) {   // quite dummy sanity check
    if (src[0] >= 190 && src[0] <= 200) { // another dummy sanity check
      send(/*@ (public) */src, dst);
      if (msg != 0) {
        compute_private_msg(msg);
        publicise(msg);
      } else
        msg = /*@ (public) */"hello world";
    } else
      result = 2;
  } else
    result = 4;

  send(msg, dst);

  compute_src(src);
  compute_dst(dst);

  send(/*@ (public) */src, dst);

  if (src[0] == 190)
    result = 2 * result + 1;

  return result;
}
```

Fig. 1. Running example: `check_and_send`

declassification operator which explicitly makes a data `public`. By default, all the values are considered private, *i.e.* non-public.

Two different properties can be studied on the function `check_and_send`:

Definition 1 (integrity). *A program fulfills the* integrity *property if no private data can be modified by an intruder which can intercept any sent message.*

Definition 2 (confidentiality). *A program fulfills the* confidentiality *property if no private data can be discovered by an intruder which can intercept any sent message.*

As for the integrity property the function `publicise` shall be a sealing primitive whereas for the confidentiality property it shall be a cyphering primitive.

Integrity is ensured by the `check_and_send` function because the only values which can be modified by an intruder are explicitly declassified or publicised. A quick similar reasoning seems to indicate that confidentiality is also ensured. Unfortunately, it is not true: the last instruction `send` emits the public data `src` which can securely be read by anyone but from which an intruder can deduce the parity of the private (*i.e.* confidential) data `result` returned by the function. This is a security leak induced by the information flow of the function.

In conclusion slicing preserving integrity can securely remove all the parts concerning the variable `result`, but slicing preserving confidentiality has to keep all the lines 1. To our knowledge there isn't any standard slicing criterion that is correct regarding the confidentiality criterion.

3 Standard Slicing Methods Do Not Preserve Security

In this section we explain the issues with the preservation of security criteria. Indeed there is no problem with the integrity criterion because preserving it is equivalent to preserve all the calls to the function `send`: only sent values are modifiable by an intruder. For this purpose, using the classical "preserve the accessibility of lines calling `send` and the value sent at these statements" criterion works fine.

The confidentiality criterion is hard to preserve because an intruder can discover confidential information through program flow. For example with the following lines of code

```
if (public_info) secret = choice1; else secret = choice2;
send(/*@ (public) */ public_info, dst);
```

the secret escapes because an intruder can *indirectly* deduce the value of `secret` from the non-confidential data `public_info` sent on the network: if the intruder intercepts the sent message, he discovers whether `public_info` is 0 or not and therefore whether the value of `secret` is `choice1` or `choice2`. So slicing according to confidentiality has to keep both lines of code, and not only the second one.

The issue remains if we simply swap the two lines of code:

```
send(/*@ (public) */ public_info, dst);
if (public_info) secret = choice1; else secret = choice2;
```

It is still possible to indirectly deduce the value of `secret` from the publicly sent data `public_info`. The confidentiality leak of our running example (Figure 1) follows this second scheme: the parity of `result` can be deduced from the last emission of `src`.

These issues bring forward a wider concern: the code that is potentially influenced by an intercepted sent value can be placed before or after — according to the program flow — the call to `send`. So security leaks may be quite difficult to identify: performing a backward or forward analysis won't work properly.

One can show analogically that confidentiality is not a compositional property: the confidentiality cannot be established for a function without knowing all its callers.

4 Security Guided Slicing: Criticality Components

Sections 2 and 3 have implicitly distinguished two different kinds of potential confidentiality leaks: *direct* and *indirect* leaks.

Direct leaks are related to program points modifying variables with values which can be *directly* deduced from a call to function `send` (the unsecured point of the program) because some modified bits may be sent by this call. For example if the function `compute_src` used in Figure 1 is defined like

```
void compute_src(int src[4]) {
  ...
  src[0] = ···;
  for(i = 1; i < 4; i++) src[i] = ···;
}
```

an intruder may deduce the values of `src[i]` ($i \in \{0, 1, 2, 3\}$) from the last call to function `send` *via* the call to function `compute_src`[1].

Indirect leaks are related to program points potentially impacted by the sent values and from which an intruder can *indirectly* deduce information from program flow. This is the problem described by all the examples given in Section 3.

In the last part of this section, we focus on a way to compute the sets of statements required to discover direct and indirect leaks from a given call to the function `send` (indeed and more generally from any statement s). We call these sets respectively the *direct and indirect criticality components* of s. In order to illustrate these computations, we consider the toy program given in Figure 2. Beforehand we introduce the necessary notions and notations.

[1] There is indeed no direct security leak at this call site because the sent value `src` is declassified. But this article is not dealing with verification but with slicing.

```
   y = ...; e = y;
A: b = ...; B: c = ...; C: x = ...; D: y = ...;
E: a = x;
F: if (c) { G: y = x; H: a = 1; }
I: z = y;
J: send(z);
   t = e;
K: if (a) { L: b = 2; }
M: d = a + b;
```

Fig. 2. Illustrating example: different kinds of dependencies

4.1 Program Dependence Graph and notations

We use $\mathcal{S}$ to denote the set of all the statements of a program. If s is a statement, we denote by $\text{reads}(s)$ the set of the memory location read by s and by $\text{writes}(s)$ the set of the memory location modified by s.

The computation of the criticality components is based on the notion of Program Dependence Graph (PDG) [9, 4] which shows dependence relations between program statements. Here we distinguish two different kinds of dependencies: *data* and *control* dependencies.

There is a *data dependency* from statement s_1 to statement s_2 according to the memory location l (a variable, a pointer access, ...) if s_2 modifies l which is used by s_1 and there is at least one path in the control flow graph from s_2 to s_1 on which l is not modified. We denote by $\text{dpds}_d(l, s)$ the set of all data dependencies of the statement s according to the memory location l. We denote by $\text{codpds}_d(l, s)$ the data *co-dependencies* of s according to l, that is $\{s \in \mathcal{S} \mid \exists s' \in \mathcal{S}, \text{dpds}_d(l, s')\}$. For example on the illustrating example (Figure 2), $\text{dpds}_d(\mathtt{x}, \mathtt{C}) = \{\mathtt{E}, \mathtt{G}\}$ and $\text{codpds}_d(\mathtt{y}, \mathtt{I}) = \{\mathtt{D}, \mathtt{G}\}$.

Informally there is a *control dependency* from the statement s_1 to the statement s_2 if s_2 conditionally decides whether s_1 is to be executed or not. A more formal definition exists [4] but it is not required in our presentation. Similarly to the data dependencies, we use the notations $\text{dpds}_c(, s)$ and $\text{codpds}_c(, s)$ for control dependencies. Following the illustrating example2, $\text{codpds}_c(, \mathtt{G}) = \{\mathtt{F}\}$.

Furthermore in order to unify the notations, we use $\text{dpds}_\delta(l, s)$ with $\delta \subseteq \{c, d\}$ to denote the set of dependencies of s according to l and regarding the kind δ of dependencies. We use the same convention for $\text{codpds}_\delta(l, s)$. We can trivially prove that $\text{dpds}_{\delta_1 \cup \delta_2}(l, s) = \text{dpds}_{\delta_1}(l, s) \cup \text{dpds}_{\delta_2}(l, s)$. The same property holds for co-dependencies. Following the illustrating example, $\text{codpds}_{cd}(\mathtt{x}, \mathtt{G}) = \{\mathtt{C}, \mathtt{F}\}$.

4.2 Computation of criticality components

The computation of criticality components is divided in two parts. Firstly we focus on the computation of *direct* criticality components. Then we introduce *indirect* criticality components. Finally we explain how slicing works with these criticality components.

Direct criticality components We first explain our algorithm for the computation of the *direct* criticality component of the statement `J` from Figure 2.

The direct criticality component, as it is informally defined in the beginning of this section, contains `J` itself and the set of statements that modify variables (and more generally memory locations) with some bits of values read in `J`. Thus it contains at least $\text{codpds}_d(l, \texttt{J})$ for each memory location l used in `J` (*i.e.* for $l \in \text{reads}(\texttt{J})$). In this case this is exactly the singleton containing `I`. `I` itself reads `y` and thus the component of `J` also contains $\text{codpds}_d(\texttt{y}, \texttt{I}) = \{\texttt{D}, \texttt{G}\}$. Similarly `G` reads `x` and therefore the component of `J` also contains $\text{codpds}_d(\texttt{x}, \texttt{G}) = \{\texttt{C}\}$. Hence the direct criticality component of `J` is $\{\texttt{J}, \texttt{I}, \texttt{D}, \texttt{G}, \texttt{C}\}$. This example shows that computing a direct criticality component of a set of statements S is an iterative process based on the data co-dependencies.

The formal definition of the direct criticality component of the set of statements S is denoted by $\upharpoonleft_d^\star S$ and defined by

$$\upharpoonleft_d S \triangleq \bigcup_{s \in S} \bigcup_{l \in \text{reads}(s)} \text{codpds}_d(l, s)$$
$$\upharpoonleft_d^\star S \triangleq \text{fix}(S \mapsto S \cup \upharpoonleft_d S)$$

where fix is the fixpoint operator on the domain $(\mathcal{P}(\mathcal{S}), \subseteq)$. The existence and uniqueness of this fixpoint operator (as well as those of the others fixpoints in this section) are easy to prove using Tarski's domain theory theorem [11] (see for example Gunter's or Winskel's books [5, 15]). Moreover on such a finite domain, fixpoints are easily computable.

Indirect criticality components In Section 3, we assert that a backward or forward analysis won't work properly. We indeed distinguish two different kinds of indirect criticality components: *backward and forward indirect criticality components* .

Backward indirect criticality components The standard scheme of code corresponding to this kind of components is described below.

```
secret = ...;
if (secret) public_info = choice1; else public_info = choice2;
send(/*@ (public) */ public_info);
```

In this scheme there is an indirect security leak because the value of `secret` escapes from the information flow: it can be deduced from the sent value `public_info` because there are control dependencies from the statements modifying `public_info` to the enclosing `if` statement.

Thus computing backward indirect criticality components is similar to computing direct criticality components but it has to take into account the control dependencies. It is indeed easy to generalise the $\upharpoonleft_d$ and $\upharpoonleft_d^\star$ operators in order to deal with any kind of dependencies and not only with the data dependencies.

Furthermore the backward indirect criticality component of the set of statements S exactly contains the statements of its non direct backward criticality component. Hence it is formally defined by

$$\upharpoonleft^{\star}_{cd} S \setminus \upharpoonleft^{\star}_{d} S.$$

In the illustrating example 2, the backward indirect criticality component of `J` is $\{\mathtt{F}, \mathtt{B}\}$.

Forward indirect criticality components The standard schemes of code corresponding to this kind of components are those of Section 3. We remind one of them below:

```
public_info = ...
send(/*@ (public) */ public_info);
if (public_info) secret = choice1; else secret = choice2;
```

In this scheme, there is an indirect security leak because the value of `secret` escapes from the information flow: its value can be deduced from the sent value `public_info` which is known thanks to the computation of the backward criticality component C of the call to `send`. One can indeed deduce information on any statement impacted by C. The impacted statements of a set of statements S are formally defined thanks to the dependencies of S in the following way:

$$\downharpoonleft S \triangleq \bigcup_{s \in S} \bigcup_{l \in \mathrm{writes}(s)} \mathrm{dpds}_{cd}(l, s)$$
$$\downharpoonleft^{\star} S \triangleq \mathrm{fix}(S \mapsto S \cup \downharpoonleft S).$$

Note that $\downharpoonleft$ and $\downharpoonleft^{\star}$ are respectively dual operators of $\upharpoonleft_{cd}$ and $\upharpoonleft^{\star}_{cd}$.

Using this operator and backward criticality components, it is easy to define the forward indirect criticality components of the set of statements S: that is the impacted statements of the backward criticality component of S which do not belong to this backward criticality component. It is formally defined by:

$$\downharpoonleft^{\star} (\upharpoonleft^{\star}_{cd} S) \setminus \upharpoonleft^{\star}_{cd} S.$$

In the illustrating example, the forward indirect criticality component of `J` is $\{\mathtt{E}, \mathtt{H}, \mathtt{K}, \mathtt{L}, \mathtt{M}\}$.

Slicing from criticality components From the above definitions, the criticality component $\Psi(S)$ of a set of statements S is computable by firstly computing its backward criticality component and secondly computing the impacted statements of the resulting sets. So Ψ is formally defined by:

$$\Psi \triangleq \downharpoonleft^{\star} \circ \upharpoonleft^{\star}_{cd}.$$

In our illustrating example, the criticality component of `J` contains all the labeled statements: unlabeled statements are sliced.

$\Psi(S)$ does not yet define a full slicer (according to the confidentiality criterion) because the resulting statements are not compilable: pieces of code like declarations leak. One can yet use standard slicing techniques on this set of statements in order to complement our methodology.

A welcome side effect of our presentation is that the notion of criticality component is related to the notion of program dependency graph and not to the *C* programming language itself: it only uses the notions of dependencies, co-dependencies, read and written memory locations and control flow graph which are very common in imperative programming language.

5 Conclusion

We have shown that standard slicing methods are not correct regarding the confidentiality criterion.

In order to solve this problem, we have presented a new slicing technique which is automatic, over-approximated-but-correct and source-to-source. It deals with security criteria, especially confidentiality, and introduces notion of *criticality component* which can be used in any imperative programming language (even if we focus on *C* in this paper).

This security slicing performs code reductions which firstly ease all auditing activities and help to focus on pertinent parts of the considered program and, secondly, may pinpoint the quality of the security of the code. Besides the code reduction may be used as a security pre-analysis to be performed before running other expensive analyses such as formal security verification.

A prototype is under development as a plug-in of the *Frama-C* (Framework for Modular Analyses of *C*) platform [2]. It uses the following native plug-ins of *Frama-C*: the value analysis, the program dependency graph and the slicer. Thanks to these plug-ins the development contains around 500 lines of code in *Objective Caml* [7] and deals with all the *C* constructs except the recursive functions calls and the signals related features. The first results are stimulating. Our main goal is the slicing of the IPsec (IP secure) [10] implementation embedded in the Linux kernel regarding the confidentiality and the integrity criteria.

References

1. Daniel J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In *CSAW '07: Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 1–10, New York, NY, USA, 2007. ACM.
2. CEA-LIST and INRIA-Futurs. Frama-C: Framework for Modular Analysis of C. `http://www.frama-c.cea.fr`.
3. 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 Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, Californie, États-Unis, 1977. ACM Press.

4. Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.
5. Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of Computing. MIT Press, 1992.
6. Gernot Heiser. Your system is secure? prove it! *USENIX ;login:*, 32(6):35–38, December 2007.
7. Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system, release 3.10*, May 2007. `http://caml.inria.fr`.
8. Bertrand Meyer. Proving pointer program properties. part 1: Context and overview. *Journal of Object Technology*, 2(2):87–108, march–april 2003. `http://www.jot.fm/issues/issue_2003_03/column8`.
9. Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, New York, NY, USA, 1984. ACM Press.
10. Stephen Kent and Karen Seo. Security Architecture for the Internet Protocol. Request for comments (rfc) 4301, Network Working Group, December 2005. `ftp://ftp.rfc-editor.org/in-notes/rfc4301.txt`.
11. Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.
12. Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
13. Mark Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, 1979.
14. Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.
15. Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, États-Unis, 1993.
16. Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.