Detection of Malicious Code in COTS Software via Certifying Compilers

Robert Charpentier and Martin Salois
{Robert.Charpentier@drev.dnd.ca, Martin.Salois@drev.dnd.ca}

Defence Research Establishment Valcartier
2459 Pie XI Blvd. North
Val Bélair, Québec, Canada
G3J 1X5

April 2000

Abstract

Information technology is more and more a vitally important underpinning to our economy and to our society. It is embedded in everyday applications and animates a wide class of systems that range from small to large and from simple to extremely sophisticated. Among the probable threats for military information systems, the presence of malicious code within COTS applications has been identified as a major risk that has not received a lot of attention. Like a virus that has infiltrated an information system during an electronic information exchange, malicious code integrated into a commercial application could remain undetected and present a major risk for the safety of information within a military system. In this paper, techniques to detect malicious code within commercial applications are reviewed. Emphasis is placed upon the certifying compiler, which enforces a formal security specification while compiling the source code. This emerging technology offers the most comprehensive and sustainable approach for large applications and for the periodic certification of upgrades.

1 Introduction

The Defence Research Establishment, Valcartier (DREV) carries out an extensive R&D program in Command and Control Information Systems (CCIS) for the Canadian Department of National Defence (DND). During the Information Warfare Workshop held in Ottawa in Oct. '96, several R&D challenges were identified and presented to DND and industry representatives [17]. Trusted software design and validation was one of the areas where additional effort was deemed necessary to meet DND needs. Of particular concern was the integration of Commercial-Off-The-Shelf (COTS) software into military information systems.

Exploiting COTS software through integration poses a distinct dilemma. On one hand, COTS software is very attractive; its use promises to reduce development time and costs. On the other hand, it introduces new risks into military information systems: hidden functionalities, trap doors, private control codes giving enhanced privileges, logical or temporal bombs [6], etc.

A feasibility study completed in 1998 indicates that a variety of software analysis techniques can be applied to the management of the risk associated with COTS software in military information systems. Among them, the exploitation of certifying compilers appears to be a very powerful technology for the efficient yet exhaustive verification of software with minimal human supervision. This paper summarises the lessons learned in the MaliCOTS project, carried out jointly by DREV and Laval University. The proposed strategy will, after successful implementation, ensure the safe integration of previously untrusted software in military information systems via certifying compilers.

2 Malicious Code

Malicious codes are fragments of programs that can affect the confidentiality, the integrity, the data and control flow, and the functionality of a system without the explicit knowledge and consent of the user. We distinguish between intentionally malicious and unintentionally malicious code. Malicious individuals who, for example, use such programs to access confidential data generally introduce the first. The second is due to inadvertent human error, especially during development of the software.

To detect malicious code in COTS software, one must be able to distinguish between its types. Starting from the taxonomy proposed by McDermott & Choi [12], a new taxonomy has been defined that is specifically intended to facilitate the detection of malicious code in COTS soft-

Paper presented at the RTO IST Symposium on "Commercial Off-the-Shelf Products in Defence Applications "The Ruthless Pursuit of COTS"", held in Brussels, Belgium, 3-5 April 2000, and published in RTO MP-48.

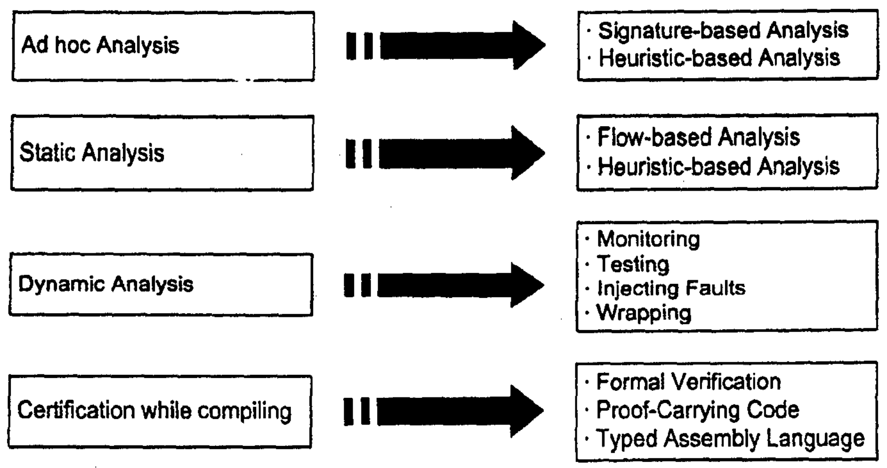

Figure 1: Potential techniques to detect malicious code in COTS software

ware [3].

One of the key concepts of the MaliCOTS project is always to refer to a security policy to distinguish an acceptable activity from a potential threat. It reflects the fact that software functionalities can never be considered malicious in and of themselves; even reformatting a disk or destroying a file are useful operations in certain circumstances; that is why such capabilities were devised and made available to system users. But in many operational contexts these functions should not be made available to end-users because of the associated risks. The most rigorous way to enforce such a policy is to formalise these constraints explicitly in a security specification based upon permissible access mechanisms. This strategy is documented in more detail in this paper.

In practice, threat that system analysts are typically concerned with are:

- the presence of trapdoors in COTS packages (as found in Unix, Windows NT & '98 [6, 9]),
- license expiration logic [16],
- hidden communications (e.g., a CD player software that is reported to send 'your listening preferences' to a distributor periodically [15]), and
- other undesirable functionalities such as those present in the flight simulator in Excel '97 and the Word '97 pinball machine [1, 2].

The next section summarises the feasibility study, completed in fall 1998, into ways to detect such malicious code.

3 Technology Options to Detect Malicious Code in COTS Software

Figure 1 identifies a variety of techniques applicable to the MaliCOTS project, in order of increasing level of complexity.

Reference [4] contains a comparative analysis of these techniques. In summary, ad hoc techniques consist of code inspection in search of a known malicious signature or its generalisation (often called a heuristic). This approach has been very successful in detecting viruses within exchanged files, but its effectiveness in detecting malicious code in large software applications is limited since a priori knowledge is needed (i.e., either signature or behaviour profiles).

Static analysis of code comes from the world of program optimisation and software analysis. It consists of examining the code (perhaps in some abstract representation) without running it. At present, static analysis is essential to COTS certification because it gives a relatively precise idea of program behaviour for all possible execution conditions. However, the technique is limited in capability, especially when source code is not available, which is typically the case for COTS. The process requires enormous human effort for very large applications [5].

Dynamic techniques examine the behaviour of the code while it is running. Such analysis is a pragmatic approach that offers short-term benefits. Many variants are available: monitoring execution, running an exhaustive suite of tests, injecting faults in critical variables or wrapping the commercial code into a software shell that detects and filters out unwanted activities. Another paper presented in this conference deals with dynamic detection and provides supplementary information [18].

Each of these techniques has its place and offers short-term solutions to the detection of malicious code in COTS software. However, they are all reactive, in the sense that they evaluate the COTS package after development, when detection is made more difficult by the lack of access to source code.

Being unsatisfied with this situation, we have searched for a truly innovative approach to COTS integration that will overcome existing difficulties with the non-availability of source code, with time-consuming manual

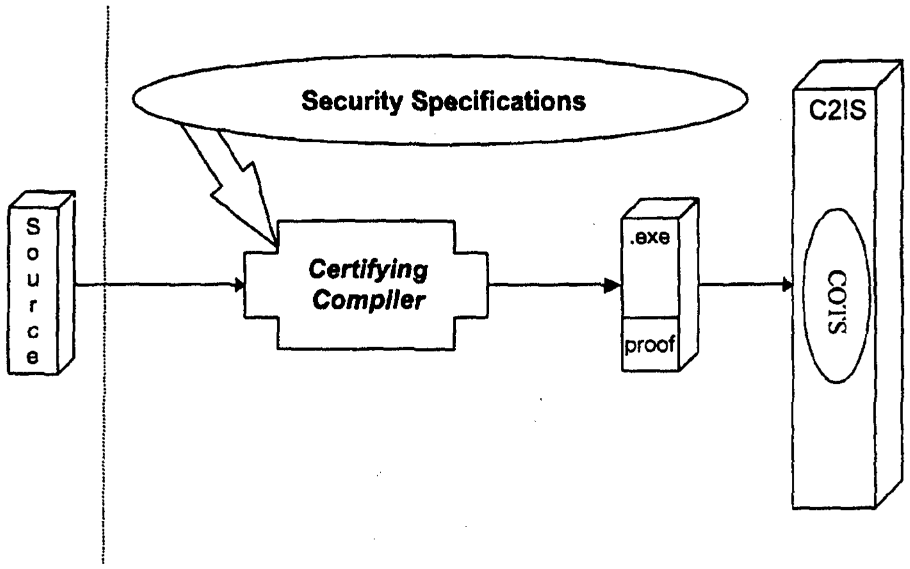

Figure 2: Certifying compiler — basic concept

inspection of software and with difficulty in ensuring the completeness of verification.

Certifying compilers have emerged as an extremely powerful technology to manage the risk associated with COTS integration. The basic idea is to put enough "intelligence" into the compiler that it will not only produce the executable code but also perform formal security verification. As shown in Figure 2, the compiler needs two inputs: the source code and the security policy. The compiler then translates the source code into the appropriate intermediate language (e.g., assembly, byte code, etc.) along with embedded security annotations.

The next section describes the concept and gives a practical feel of its capabilities in our particular context.

4 Certifying Compilers; Concept & Practice

4.1 Concept

As in human health, prevention is certainly the best cure. So it is worthwhile from a security standpoint to elaborate methodologies that guarantee that COTS software products are free of any malicious code from the start. In order to do that, we propose the inclusion of intelligence in the compiler to allow enforcement of a security specification while compiling.

Figure 3 illustrates the most general scheme to produce trusted software while compiling. The first step consists of compiling the source code and introducing static annotations in the object file (i.e., byte code for JAVA, assembly language or other intermediate language). It is a rather simple and mechanical process to introduce the annotations. Secondly, the annotated code is submitted to a verifier (or a verifying linker) that enforces a formally expressed security specification. By doing so, the final executable application can be assembled safely and sealed with a security tag before integration into a critical information system.

This is a very flexible approach. Not only can the annotations be produced rapidly and independently of the final integration but also different local security policies can be enforced in different parts of an organisation on a single annotated component. Another great advantage of this approach is that there is no need for the software integrator to have access to the source code. The only requirement for the software producer is to adopt an annotation structure that the integrator can recognise and verify for correctness. This key feature protects the intellectual property of software producers.

The second step of the process (verification) starts with a comparison of the annotations with the object code. Any anomalies in the compliance of the code with the annotations can easily be flagged for further investigation. In other words, if the code is modified after it was annotated, or if the annotations are changed without any code modification, the verifier will rapidly detect it. The only component that one must trust in this system is the verifier itself; there is no need for trustworthiness in the code producer, the annotating compiler or the transmission channel up to the verifier. This is a very important feature for security architects, who may deal with the trustworthiness of only one component, the verifier.

4.2 Annotation Structures

So far we have not described the content and the structure of the annotations the compiler produces. Many options exist, each with its advantages and disadvantages. In

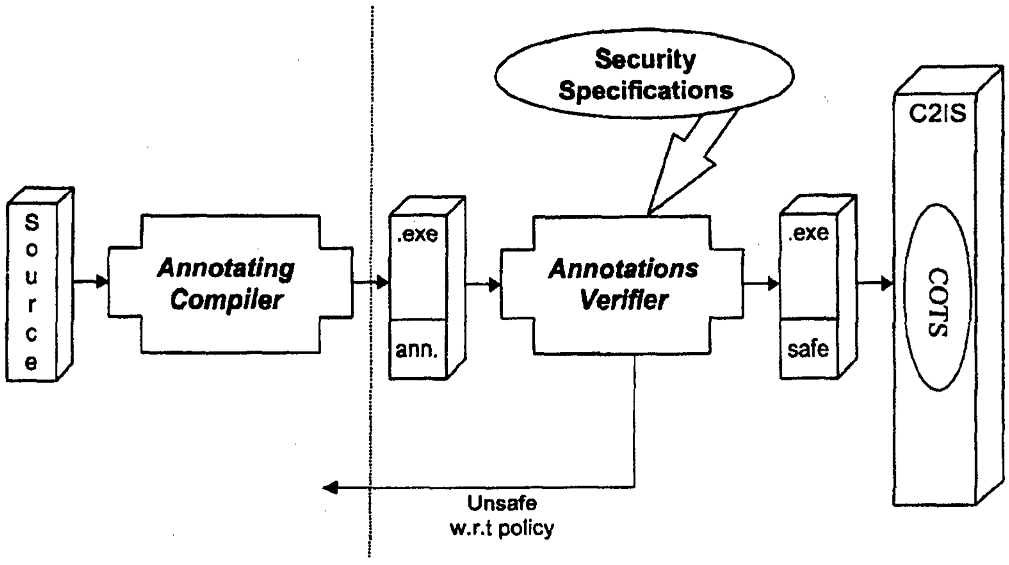

Figure 3: Certifying compiler — generalized concept

the MaliCOTS projects, we examined three possibilities closely:

- PCC (Proof Carrying Code), developed under the leadership of Peter Lee and George Necula at CMU (Carnegie Mellon) and at Berkeley University;
- ECC (Efficient Code Certification), led by Dexter Kozen from Cornell; and
- TAL (Typed Assembly Language), designed by Greg Morissett at Cornell University.

PCC is a technique to ensure the safe execution of untrusted mobile code. When code is transferred between a client and a producer, the producer must append to the code a formal proof that it is consistent with some shared security policy. The client can easily check the proof by using a simple and easy-to-trust proof checker. PCC is a very comprehensive and secure approach [14].

ECC was designed to be a much lighter solution to code certification. The annotations contain structured information that qualifies the safety of the code. It was designed for efficiency and performance, sacrificing some of the rigour of other approaches [10].

TAL proposes to introduce "type"-typing information into the code. Basically, software types are static descriptors of logical entities (e.g., variables, constants, character strings...) and of how they are used in the code. These annotations are light and informative and can easily be produced and managed within a comprehensive security policy [13].

TAL was selected as the technology of choice for the detection of malicious code in COTS software. Type annotations provide an automatic way to verify that a program will not violate safety properties and, potentially, high-level security requirements. At this time, TAL can handle:

- control flow safety (i.e., programs cannot jump to code that was not verified and stack preservation is enforced),
- memory safety (i.e., access to initialised memory locations and array bounds checking) and
- type safety (i.e., the compatibility of types in operations).

Complementary annotations in the "ECC style" will be considered later in the MaliCOTS project if they are needed.

In summary, type annotations are static approximations of the behaviour of the program. Essentially, they correspond to typing preconditions on code labels. Before transferring control to any label, the register, stack and relevant variables must contain values of the types specified. The type-checker matches each instruction operand against these constraints to ensure that they do not violate safety properties.

4.3 Example

To illustrate the concept of annotations, we will now examine a simple program written in C (Code Excerpt 1) and compile it to assembly language with annotations (TALx86 code) as shown in Code Excerpt 2, where annotations appear in bold. An expression such as "eax: B4" indicates that the register "eax" must contain four bytes if the following instruction is to be executed. Inference rules are used to verify formally that all conditions are met before the activation of a given operand (e.g., an arithmetic operation or a call procedure like those shown in Figure 4).

Code Excerpt 1: Sample C code

```
#define TABLEAU 100
unsigned int premiersTABLEAU;

int estPremier( int nombre, int compte )
{
   int i = 0;
   int iPremier = 1; //sans preuve du contraire, c'est un nombre premier

   for (i = 0; i < compte; i++)
   {
      if ( nombre % premiersi == 0 )
      {
         iPremier = 0;
         break;
      }
   }

   return iPremier;
}
```

$$(ArithBin) \quad \frac{\varepsilon \vdash op_1 : B4 \qquad \varepsilon \vdash op_2 : B4 \qquad \varepsilon \vdash ValidBinops(op_1, op_2) \qquad \varepsilon \vdash Writeable(op_1)}{\varepsilon \vdash arithbin\ op_1, op_2 : \varepsilon}$$

$$(Call) \quad \frac{\varepsilon \vdash cop : \{g_1\} \qquad \varepsilon \vdash g_1(\mathtt{esp}) = sptr\{g_2 :: c'\} \qquad \varepsilon \vdash \varepsilon.\gamma[\mathtt{esp} : sptr\{g_2 :: (\varepsilon.\gamma(\mathtt{esp}))\}] \preceq g_1}{\varepsilon \vdash \mathtt{call}\ cop : \varepsilon[\gamma : g_2]}$$

Figure 4: Two inference rules enforcing annotation checking in TAL

As part of the MaliCOTS project, we are developing an ANSI C compiler that will produce assembly language for x86 processors along with the corresponding TAL annotations. Our compiler is based on LCC (Lean retargetable C Compiler); a public-domain compiler that is well documented and for which source code is available [7]. A beta version of our TalCC compiler is available for government release, to allow a broader community to become familiar with the annotation technology. More information can be obtained from the authors of this paper.

For next year, we are planning the development of a JAVA annotating compiler that exploits the same annotation structure as TalCC. It will probably be based on JIKES, an IBM shareware compiler that is part of Linux packages. Emerging commercial products will also be considered ([11]).

5 Discussion & Conclusion

In view of budget reductions and decreasing human resources, integration of COTS software appears to be the only sustainable approach for Canadian DND [8]. At the present time, system analysts have only such labour-intensive techniques as static and dynamic verification to certify COTS software. It is expected that these techniques will remain useful (and mandatory, in many instances) for the certification of COTS packages. The MaliCOTS team values them greatly and attempts to integrate them into a common framework.

However, it is evident that more efficient and less time-consuming techniques are needed to handle COTS software, especially when periodic upgrades must be certified and when security policies must be met that vary significantly throughout an organisation.

Certifying compiler is a powerful enabling technology to meet this challenge. By formally specifying local security policies and by annotating an intermediate form of the code, the whole process is brought under control. Marginally acceptable functionalities and suspicious code segments may require later manual inspection, but the software core can be certified autonomously by the verifier.

This approach is also general enough to contribute to other kinds of certification, including interoperability compliance, reuse policy, maintainability specifications, etc., which are not examined by the MaliCOTS team at this time. Once these additional policies are expressed formally, simply passing the verifier over the annotated code would enforce them. Even though they are simple and compact, type annotations are very expressive. Our R&D on the detection of malicious code confirm that they have a strong potential for structuring and normalising the integration of COTS software into critical systems.

The expected benefits of certifying compilation are extensive and far-reaching. We hope that this paper will create enough interest in the technology that international collaboration can be organised to explore this ambitious certification paradigm more fully.

References

[1] T. E. E. Archive. Excel 97 Flight to Credits. `http://www.eeggs.com/items/718.html`.

[2] T. E. E. Archive. Pinball in Word 97. `http://www.eeggs.com/items/763.html`.

[3] J. Bergeron, M. Debbabi, J. Desharnais, B. Ktari, M. Salois, and N. Tawbi. Skeleton of a Taxonomy for Malicious Code. Technical report, DREV, Nov. 1998.

[4] J. Bergeron, M. Debbabi, J. Desharnais, B. Ktari, M. Salois, N. Tawbi, R. Charpentier, and M. Patry. Detection of Malicious Code in COTS Software : a Short Survey. In *First International Software Assurance Certification Conference (ISACC'99)*, Washington D.C., Mar. 1999. Section C1.

[5] J. Bergeron, M. Debbabi, M. M. Erhioui, and B. Ktari. Static Analysis of Binary Code to Isolate Malicious Behaviors. In 4th *International Workshop on Enterprise Security*, Stanford University, California, USA, June 1999. IEEE Computer Society Press.

[6] J. T. Egan. Information Security Threats to Software Development. In *Software Technology Conference*, USA, Apr. 1997.

[7] C. Fraser and D. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995. ISBN 0-8053-1670-1, `q. http://www.cs.princeton.edu/software/lcc/`.

[8] C. M. Hanrahan. Changing the Culture (COTS vs. Development). In *COTS Software Seminar*, Ottawa, Feb. 1998.

[9] G. Hoglund. A *REAL* NT Rootkit, Patching the NT Kernel. *Phrack Magazine*, 9(55), Sept. 1999.

[10] D. Kozen. Efficient Code Certification (ECC). Technical Report TR98-1661, Cornell University, Jan. 1998.

[11] P. Lee and G. C. Necula. Cedilla systems inc. `http://www.cedillasystems.com/`.

[12] J. P. McDermott and W. S. Choi. Taxonomy of Computer Program Security Flaws. *ACM Computing Surveys*, 26(3):211–254, Sept. 1994.

[13] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A Realistic Typed Assembly Language. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, May 1999.

[14] G. C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 1–14, Paris, France, Jan. 1997. `http://www.cs.cmu.edu/~necula/pop197.ps.gz`.

[15] NTSecurity.net. Leap of Faith Now Required for Real Networks? `http://www.ntsecurity.net/forums/2cents/news.asp?IDF=173&TB=news`, Nov. 1999.

[16] Quarterdeck. Aids Information Kit Trojan. `http://www.quarterdeck.com/quarc/00000/00000030.htm`, June 1994.

[17] R. Roy, editor. *Strategic Investment Workshop Proceedings*. CRAD, Oct. 1996. Canadian Eyes Only.

[18] M. Salois and R. Charpentier. Dynamic Detection of Malicious Code in COTS Sofware. In *Commercial Off-The-Shelf Products in Defence Applications "The Ruthless Pursuit of COTS"*, Neuilly-sur-Seine Cedex, France, Apr. 2000. NATO, RTO.

6 Acknowledgements

The authors wish to thank the MaliCOTS collaborators who contribute to this ambitious research effort. Currently, 12 Laval University graduate students and 4 professors are involved. Special thanks must be addressed to Dr Mourad Debbabi, who has led this team with competence and dedication.

Code Excerpt 2: Corresponding assembly language with annotations in TAL

```
_estPremier:
 LABELTYPE < All[ s1: Ts , n1: Sint ].{ ESP: sptr[S(0)] ({ ESP: sptr[S(0)] B4::B4::s1 ,
   EBP: sptr[S(n1)] s1 , EAX: B4 })^x::B4::B4::s1 , EBP: sptr[S(n1)] s1 } >
 push ebx
 push esi
 push edi
 enter 8,0
 L8:
 LABELTYPE < All[ s1: Ts , n1: Sint ].{ ESP: sptr[S(0)] B4^u::B4^u::E4^x::B4^x::B4^x::B4^x::
   ({ ESP: sptr[S(0)] B4::B4::s1 , EBP: sptr[S(n1)] s1 , EAX: B4 })^x::B4::B4::s1 , EBP:
   sptr[S(-8)] B4^u::B4^u::B4^x::B4^x::B4^x::B4^x::({ ESP: sptr[S(0)] B4::B4::s1 , EBP:
   sptr[S(n1)] s1 , EAX: B4 })^x::B4::B4::s1 } >
 mov dword ptr (-4) [ebp] ,0
 mov dword ptr (-8) [ebp] ,1
 mov dword ptr (-4) [ebp] ,0
 jmp tapp( L5, < s1, n1 > )
 L2:
 LABELTYPE < All[ s1: Ts , n1: Sint ].{ ESP: sptr[S(0)] B4^rw::B4^rw::B4^x::B4^x::B4^x::B4^x::
   ({ ESP: sptr[S(0)] B4::B4::s1 , EBP: sptr[S(n1)] s1 , EAX: B4 })^x::B4::B4::s1 , EBP:
   sptr[S(-8)] B4^rw::B4^rw::B4^x::B4^x::B4^x::B4^x::({ ESP: sptr[S(0)] B4::B4::s1 , EBP:
   sptr[S(n1)] s1 , EAX: B4 })^x::B4::B4::s1 } >
 mov edi,dword ptr (20) [ebp]
 mov eax,edi
 mov edi,dword ptr (-4) [ebp]
 mov edi,dword ptr (_premiers) [edi*4]
 xor edx,edx
 div edi
 cmp edx,0
 jne tapp( L6, < s1, n1 > )
 L9:
 LABELTYPE < All[ s1: Ts , n1: Sint ].{ ESP: sptr[S(0)] B4^rw::B4^rw::B4^x::B4^x::B4^x::B4^x::
   ({ ESP: sptr[S(0)] B4::B4::s1 , EBP: sptr[S(n1)] s1 , EAX: B4 })^x::B4::B4::s1 , EBP:
   sptr[S(-8)] B4^rw::B4^rw::B4^x::B4^x::B4^x::B4^x::({ ESP: sptr[S(0)] B4::B4::s1 , EBP:
   sptr[S(n1)] s1 , EAX: B4 })^x::B4::B4::s1 } >
 mov dword ptr (-8) [ebp] ,0
 jmp tapp( L4, < s1, n1 > )
 L6:
 LABELTYPE < All[ s1: Ts , n1: Sint ].{ ESP: sptr[S(0)] B4^rw::B4^rw::B4^x::B4^x::B4^x::B4^x::
   ({ ESP: sptr[S(0)] B4::B4::s1 , EBP: sptr[S(n1)] s1 , EAX: B4 })^x::B4::B4::s1 , EBP:
   sptr[S(-8)] B4^rw::B4^rw::B4^x::B4^x::B4^x::B4^x::({ ESP: sptr[S(0)] B4::B4::s1 , EBP:
   sptr[S(n1)] s1 , EAX: B4 })^x::B4::B4::s1 } >
 L3:
 LABELTYPE < All[ s1: Ts , n1: Sint ].{ ESP: sptr[S(0)] B4^rw::B4^rw::B4^x::B4^x::B4^x::B4^x::
   ({ ESP: sptr[S(0)] B4::B4::s1 , EBP: sptr[S(n1)] s1 , EAX: B4 })^x::B4::B4::s1 , EBP:
   sptr[S(-8)] B4^rw::B4^rw::B4^x::B4^x::B4^x::B4^x::({ ESP: sptr[S(0)] B4::B4::s1 , EBP:
   sptr[S(n1)] s1 , EAX: B4 })^x::B4::B4::s1 } >
 inc dword ptr (-4) [ebp]
 L5:
 LABELTYPE < All[ s1: Ts , n1: Sint ].{ ESP: sptr[S(0)] B4^rw::B4^rw::B4^x::B4^x::B4^x::B4^x::
   ({ ESP: sptr[S(0)] B4::B4::s1 , EBP: sptr[S(n1)] s1 , EAX: B4 })^x::B4::B4::s1 , EBP:
   sptr[S(-8)] B4^rw::B4^rw::B4^x::B4^x::B4^x::B4^x::({ ESP: sptr[S(0)] B4::B4::s1 , EBP:
   sptr[S(n1)] s1 , EAX: B4 })^x::B4::B4::s1 } >
 mov edi,dword ptr (24) [ebp]
 cmp dword ptr (-4) [ebp],edi
 jl tapp( L2, < s1, n1 > )
 ...
```