Taint Analysis of Security Code in the KLEE Symbolic Execution Engine

Ricardo Corin[1,2] and Felipe Andrés Manzano[1,3]

[1] FaMAF, UNC, Argentina
[2] CONICET
[3] Binamuse, Inc.
rcorin@famaf.unc.edu.ar, feliam@binamuse.com

Abstract. We analyse the security of code by extending the KLEE symbolic execution engine with a tainting mechanism that tracks information flows of data. We consider both simple flows from direct assignment operations, and (more subtle) indirect flows inferred from the control flow. Our mechanism prevents overtainting by using a region-based static analysis provided by LLVM, the compiler infrastructure machine on which KLEE runs. We rigorously define taint propagation in a formal LLVM intermediate representation semantics, and show the correctness of our method. We illustrate the mechanism with several examples, showing how we use tainting to prove confidentiality and integrity properties.

1 Introduction

Analysis methods based on *symbolic execution* (developed initially by King [8]) have been proved to scale very well to real life code. For instance, KLEE [3], a symbolic execution engine running on top of the LLVM low-level virtual machine [9], has been used to identify subtle bugs in the popular GNU COREUTILS library, covering over 430K lines of C code.

In symbolic execution, the program is dynamically explored through all its branches looking for implementation bugs like memory manipulation errors. In order to avoid trying the entire (arbitrarily large) input space, program inputs are assumed to be symbolic variables that remain uninstantiated (but become constrained) at execution time.

Unfortunately, existing symbolic execution tools can't deal with code that uses cryptography: the search space blows up when exploring the insides of such functions, as they are specifically designed to avoid being invertible (e.g., hash or encryption operations), and hence the underlying constraint solver of the symbolic execution engine (e.g., STP [6] for KLEE) is unable to find suitable inputs in a reasonable time.

In order to cope with this problem, recent work [5] extends KLEE by introducing "symbolic" functions that replace concrete ones (e.g., a symbolic encryption function symbol replacing a OpenSSL's AES implementation), and prevent their exploration. In order to specify the behaviour of symbolic functions and allow the analysis to progress, symbolic functions are endowed with rewriting rules

T.W. Chim and T.H. Yuen (Eds.): ICICS 2012, LNCS 7618, pp. 264–275, 2012.

that detail abstractly their functional properties (e.g., that decryption inverts encryption). The advantage is that these symbolic functions can be efficiently implemented using lookup tables, enabling the symbolic execution of the whole protocol.

```
unsigned char K[]
          = "SECRETSECRET";

void
oracle(){
int i;
unsigned char IV[256];
unsigned char C[256];
unsigned char P[256];

read(IV, 256);
read(C, 256);

decrypt(P, C, K);

//XOR with previous block/IV
for (i=0;i<256;i++)
        P[i] ^= IV[i];
//Check padding..
if (is_valid(P))
   write(``valid'',5);
else
   write(``invalid'',7);
}
```

```
int is_valid(unsigned char * P){
    int i;
    //valid paddings in [1,256]
    for (i=256-P[255];i<255;i++)
        //all pads = pad length
        if ( P[255] != P[i] )
            return 0;
    return P[255] != 0;
}
```

Fig. 1. Is this code secure?

Taint analysis [12] is a powerful method for discovering security violations. The analysis is used typically to identify dangerous flows from untrusted inputs into sensitive destinations, in order to detect, for instance, code or SQL injection attacks. This is an *integrity* property that tells whether untrusted values can reach and modify trusted placeholders. One may also be interested in the dual property of *confidentiality*, i.e., whether sensitive values can leak to untrusted sinks (e.g., whether secret information is disclosed to the network). We rely on tainting to formally justify the usage of symbolic function abstractions for replacing concrete cryptographic primitives. In order to show it is safe to perform such an abstraction, we need to reason about which information needs to be kept secret. For instance, consider an encrypted message that is sent on the network; the encryption key has been established off-band and is meant to be secret. We can only replace this encryption by a symbolic (black box) message that is totally

opaque to an attacker when the key is actually verifiably secret to the attacker. That is, we can trust an eavesdropped message will not be decrypted (nor any information will deduced from it) by an attacker only if no information about the encryption key was (inadvertently) leaked by the program itself.

Let us illustrate the kind of code we wish to analyse. Consider the *oracle* C function of Figure 1. It represents a *last word oracle* used in the classic padding oracle attack [13,10]. Two 256-byte long blocks are input in lines 11 and 12, the initialization vector IV and the ciphertext C. Line 14 decrypts C into P using secret key K. Then, lines 17 and 18 xor the result with IV (since this is using CBC encryption mode for block ciphers). Finally, the padding is checked with function is_valid() in line 20. This function, shown in the right hand-side of Figure 1, checks the padding method is valid (using PKCS#5), i.e., that the last bytes are either 1, or 22, 333, and so on. This code illustrates the confidentiality concerns we are interested in analysing: Can an attacker obtain information about P from observing just the output?

As shown in [13], an attacker providing a random IV and observing the answer (that is, "valid" or "invalid" depending on the output of is_valid(P)) can infer the last byte of P. In this paper, we develop a mechanism to detect subtle leaks of information of this class. Briefly, in our analysis, the decrypted P of line 13 is secret and thus marked high (by specification), with security level H. Then, at runtime, the analysis detects information being output under a high guard (i.e., the conditional in line 19 of the result of function is_valid()). At that point our analysis would issue a security warning.

The above example illustrates our interest in detecting *all* potential leaks, including partial information. So even a 1-bit leak of P constitutes a valid attack in our setting. We aim at formal results, thus we formalize the LLVM semantics on which KLEE runs. Previous work describes the standard LLVM semantics [5], and we extend it here to model taint propagation. This enables us to show formally that tainting works as desired. More precisely, our contributions are as follows:

- We illustrate, via examples, the challenges in implementing different tainting mechanisms in the LLVM virtual machine (Section 2).
- We define three LLVM semantics to model taint propagation, each one more precise than the previous (Section 3):
  1. A basic tainting semantics for modelling direct, assignment-based flows.
  2. A more advanced tainting semantics for both direct and indirect flows arising from branching operations.
  3. A region-based tainting semantics that prevents overtainting, and is thus more precise than the previous case while still correct.

  We show security for both (2) and (3) above (Theorems 1 and 2, respectively).

Even though our development of tainting is aimed towards analysing cryptographic protocol implementations, it can of course be used in analysing regular, non-protocol code, in order to detect dangerous flows of data.

Related Work. To our knowledge, this is the first tainting/information flow approach specific for the LLVM virtual machine and KLEE, which combines both a working prototype and rigorous semantics with formal security results. However, of course there exist lots of work for tracking information in programming languages. First, more applied taint analyses [2]: tainting has been used for unknown vulnerability detection, automatic input filter generation, malware analysis, and test case generation (see the survey in [12] and the references therein). Second, more theoretic information flow works [11], both for static and dynamic settings, and for higher and lower level languages.

The first work we are aware on defining formal semantics for LLVM is [5]. There is more recent work [14] that also gives semantics, and focuses in mechanized formalizations of LLVM for proving intermediate optimizations correct in the Coq theorem prover.

2 Information Flow and Tainting in the LLVM

In order to understand the semantic rules needed to implement tainting in Section 3, in this section we consider some simple examples that illustrate the kind of issues we run into when dealing with tainting. Our examples are purposely simple, since we work at the level of LLVM IR (intermediate representation) code, which is considerably more verbose than C.

LLVM IR code is organized as a collection of function definitions, each one containing a sequence *basic blocks*. Each basic block is tagged with an entry label, to which other blocks can jump into. Basic blocks typically end when control needs to be transferred elsewhere. LLVM provides "local variables" (registers), which are identified by starting with a '%' character. Registers are used thoroughly, since they are often needed by the compilers in order to generate code that respects static single assignment (SSA), a property that simplifies LLVM's static analysis and optimizations (e.g., constant propagation). The complete LLVM language contains many instructions; in this paper we use and illustrate the main ones, like arithmetic, branching, routine call and return, and memory manipulation operations. The complete list is available elsewhere [9].

We assume given an arbitrary set of *taint levels* that we use to taint variables, be them registers or memory cells. Initially, the memory (which contains data as well as the executable code) is untainted. Tainted data is introduced from the external environment of the program, and once inside the program starts propagating through variables and memory cells during execution. We allow different sources of data, which may be potentially tainted with different levels. We model the sources as files (in the UNIX sense, so that files can also be IO devices and network connections) that the code reads from and writes to. We then assign taint levels to files. Taint levels are (partially) ordered. For simplicity, and without loss of generality, we present our examples assuming just two levels: L (for low) and H (for high), with L $<$ H. In these examples, the experiment we run is as follows: we assume H as the taint level for the inputs from the standard input (0 in POSIX systems), and then check that the taint level of data written

to the standard output (1 in POSIX systems) is L. If we ever see an output H, we declare that there is a dangerous flow, and conclude the code is insecure.

Direct Flows. Figure 2 shows the simplest form of information flow: an assignment that transfers taint from a source variable to another variable. On the left we see C code and on the right we show equivalent LLVM code. The C code declares two variables of one byte, of type char, named *input* and *output*. We then input a byte from the the standard input (which we assign taint level H), using function *read*. The input byte is saved on *input*. Then we assign *input* to *output*, and finally send *output* to the standard output using function *write*. It is clear that data flows from variable *input* to *output*, hence, there is a (dangerous) flow, since H data is being leaked.

On the right hand side of Figure 2 we see the LLVM IR code, which it's more verbose and complex; it uses registers as well as memory accesses, and types are explicit ($i8$ for a byte, $i32$ for 4-byte integer; the types for constants 0 and 1 are ommitted).

C variables *input* and *output* correspond to different memory locations reserved by the LLVM operation *alloca* (lines 1–2). Pointers $\%input\_p$ and $\%output$ (respectivelly) reference both variables. Line 3 reads a byte into memory location pointed by $\%input\_p$, and line 4 loads that char into register $\%input$. The relevant flow occurs at line 5, where local register $\%input$ is assigned a value loaded from memory using operation *load*. The memory pointer used, $\%input\_p$, is used in function *read* with file descriptor 1 (which has taint level H). Line 6 uses another memory operation, *store*, to save in memory the value of $\%input$ into memory location $\%output\_p$. This example shows the need to propagate taint levels both from and back memory cells into registers, something we address in the semantics developed in next section.

Being able to capture direct flows is already quite useful, and many taint techniques do just that [12], since each flow may potentially lead to a dangerous bug.

Indirect Flows. Figure 3 shows a more subtle flow. As we can see in the C code on the left, a H variable *input* is used to switch and assign to variable *output* different values. The effect is that at the end of execution, *output* holds the value of *input*, even though there is no direct flow from *input* to *output*. So tainting *input* would not directly taint *output*. This is a classical indirect flow arising from the control flow: *input* controls a conditional (the switch) in the code that has an impact on *output*. On the right hand side of Figure 3 we see the equivalent LLVM IR code. LLVM has a primitive switch operation as well (see line 5), so the mapping is quite direct, as each branch is implemented via jumps to entry labels of the different basic blocks (e.g., $bb0$ in line 11).

The standard way of detecting these flows is to taint the control flow of the program under execution, so any following operation and its resulting memory or register modifications gets tainted with the control flow taint. In this case, the switch statement of line 5 causes the control flow to be tainted with H since we have a condition on variable *input* which is itself H. Then all following

instructions inherit the taint level H, effectively tainting the result of any instruction, including the assignment to *output*. This works, but has the potential problem of overtainting, since the H level is now carried on forever, unless one can somehow turn it off at some point (see Section 3). Nevertheless, thanks to the static analysis facilities provided by the LLVM framework, we will be able to compute regions where each branch under the influence of the switch terminates and merges into a common point (*bb*256 at line 26 in the example); this information is going to be used in Section 3 to prevent overtainting, by knowing when to stop carrying the control flow taint introduced in H branches.

```
1  char input;
2  char output;
3  read(H, &input, 1);
4  output = input;
5  write(L,&output,1);
```

```
1  %input_p = alloca i8
2  %output_p = alloca i8
3  call i32 @read( 1, i8* %input_p, 1)
4  %input = load i8* %input_p
5  store i8 %input, i8* %output_p
6  call i32 @write( 0, i8* %output_p, 1)
```

Fig. 2. Direct flow example: C code (left), LLVM IR code (right)

```
1  char input;
2  char output;
3  read(H, &input, 1);
4
5  switch(input){
6     case 0:
7          output = 0;
8     case 1:
9          output = 1;
10    ...
11    case 255:
12         output = 255;
13    }
14 write(L, output, 1);
```

```
1  %input_p = alloca i8
2  %output_p = alloca i8
3  call i32 @read( 0, i8* %input_p, 1)
4  %input = load i8* %input_p, align 1
5  switch i8 %input, label %bb256 [
6            i8 0, label %bb0
7            ...
8            i8 255, label %bb255 ]
9  bb0:
10 store i8 0, i8* %output_p
11 br label %bb256
12 ...
13 bb255:
14 store i8 255, i8* %output_p
15 br label %bb256
16 bb256:
17 call i32 @write( 1, i8* %output_p, 1)
```

Fig. 3. Indirect flow via conditionals example: C code (left), LLVM IR code (right)

Pointer Arithmetic and Memory. Figure 4 shows another subtle situation. We only show the C code for simplicity. This code copies the value of variable *input* into *output* in a special way: it initializes an array with zeroes, and then stores a 1 into the position given by *input*. Then the array is traversed until a 1 is found; while it is not 1, *output* is incremented. At the end of the loop, *output* holds the value of *input*, and this constitutes a dangerous flow since *output* is sent to the environment in line 7. Unfortunately the prevention mechanism we

hinted above to deal with indirect flows in the last subsection does not work here: marking the control as H at the branch does not help in identifying this flow, as execution is not under a branch depending on *input*: the loop uses $array[0] \ldots array[input - 1]$ but never reaches $array[input]$.

The problem here is in line 5: since we have a H index (*input*) accessing the array, the whole array is potentially H. If we could mark all the array as H, *output* would become tainted immediately entering the loop, and then the problem would disappear, as $array[0]$ would already be H. At the C code, we could mark the whole array as H if one element has been marked H. Unfortunately at the LLVM IR level we do not have arrays anymore, only memory cells. The conservative decision we make in our semantics in next section is to mark the *whole* memory as H, for this particular case. This can be made more precise in the future, although it is enough for our current needs. (For instance, memory reserved afterwards is unaffected.)

3 Taint Semantics

Our semantics is an extension of earlier work [5]. Besides that work, one can also refer to the original (informal) semantics of LLVM [9].

Semantic rules describe formally how an LLVM machine executes, i.e., how it evolves from state to state depending on the current instruction. State is represented by a tuple $\langle pc, \mathcal{M}, \mathcal{G}, fs \rangle$, where pc is the program counter, $\mathcal{M}$ the memory, $\mathcal{G}$ the global identifiers, and fs the stack of activation frames.

The specific details on how LLVM works for each instruction are not crucial for the understanding of this paper, though, since our addition of tainting does not change the semantics of [5], only builds on it.

Direct Flow Semantics. We tag LLVM registers and memory cells with taint levels H and L, and use metavariable tl to range through them. Given two levels tl_1 and tl_2 we use lattice join operation $\vee$ that operates as usual: $tl_1 \vee tl_2 = L$ when $tl_1 = tl_2 = L$, and $tl_1 \vee tl_2 = H$ otherwise.

To illustrate the semantics, we show rule ADD for arithmetic addition:

$$\frac{op_{\mathcal{M}}(pc) = id = \texttt{ add } t\ op_1, op_2 \qquad v(op_1, \mathcal{L}) = (tl_1, t, v_{op_1}) v(op_2, \mathcal{L}) = (tl_2, t, v_{op_2})}{\begin{array}{c}\langle pc, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs \rangle \longrightarrow \\ \langle nxt_{\mathcal{M}}(pc), \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}\{id \rightarrow (tl_1 \vee tl_2, t, x_{op_1} +_t x_{op_2})\}, ret, \mathcal{A}) :: fs \rangle\end{array}} \texttt{ADD}$$

It starts by looking up the instruction pointed by the program counter pc, using auxiliary function $op_{\mathcal{M}}(pc)$ (in this rule, $op_{\mathcal{M}}(pc) = id = \texttt{ add } t\ op_1, op_2$). This gives two operands op_1 and op_2, of type t. The actual values of op_1 and op_2 are looked up using auxiliary function $v()$ (the value can be either a constant with taint level L, or a local binding in $\mathcal{L}$, or a global value in $\mathcal{G}$). For op_1, for instance, we have $v(op_1, \mathcal{L}) = (tl_1, t, v_{op_1})$. Here we get a triple indicating the taint level tl_1, the type t, and finally the value v_{op_1}. (Similarly for op_2.) We can then update the context with the new value for identifier id with triple $(tl_1 \vee tl_2, t, v_{op_1} +_t v_{op_2})$ The

remaining arithmetic rules (for subtraction, multiplication, bit manipulation, and so on) are similar.

This semantics is concrete: bytes input are concrete bytes, and conditions in branching rules are deterministically evaluated. A symbolic semantics changes input values to symbolic variables, and branching rules may depend on actual assignments to the symbolic variables. A symbolic semantics of LLVM is provided in [5]; we could do it here, although for our purposes is not needed: the changes we do for tainting are completely orthogonal (under the assumption that the *pc* is always concrete, that is, there's no dynamic code).

```
1  char array[256]={0};//256 zeros
2  char input;
3  char output;
4  read(H, &input, 1);//read a char from a H file
5  array[input]=1;
6  for(output=0; !array[output]; output++); //Count zeros
7  write(L,output,1);//write output to a L file
```

Fig. 4. Indirect flow via array indexing example (C code)

Indirect Flow Semantics. In order to account for indirect flows, we need to modify contexts and carry on a taint level of the execution. We call this the taint level of the program counter, noted by *tpc*, and add it to regular contexts: $\langle pc, tpc, \mathcal{M}, \mathcal{G}, fs\rangle$. Initially *tpc* is L, and it evolves when H conditions are evaluated in branches. The rules of interest are conditionals (BRT) and (BRF); next we show rule (BRT):

$$\frac{op_{\mathcal{M}}(pc) = \texttt{br}\ c\ \texttt{labelt}\ l_1\ \texttt{labelt}\ l_2 \qquad v(c,\mathcal{L}) = (ts, \mathbf{i1}, 1)}{\langle pc, tpc, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs\rangle \longrightarrow \langle l_1, tpc \vee tl, \mathcal{M}, \mathcal{G}, (rslt, \mathcal{L}, ret, \mathcal{A}) :: fs\rangle}\texttt{BRT}$$

Here, we can see that the condition c is evaluated into a taint level ts, and this taint level is used to update the taint level of the program counter, which becomes $tpc \vee tl$. This is analogous in rule (BRF).

We conclude this subsection by noting that all regular direct-flow semantics have the control flow taint added in their results; for instance, the new rule ADD includes the taint level control flow *tpc* added to the result taint level of *id*, assigning to it the taint level of $tpc \vee tl_1 \vee tl_2$.

Analysis Methodology. Armed with any of the above semantics, we can check integrity and confidentiality by querying the program at specific places to check the taint level of certain variables of interest.

In the following we focus on confidentiality (integrity is analogous). We let an execution trace $tr(h_1, \ldots, h_n) \rightarrow o_1, \ldots, o_k$ stand for a chain of semantic rules from the initial context (as defined in [5]), with n input READ rules from H applied assigning byte h_i, in order. Analogously, the trace contains k WRITE

rules, and each output value is o_i, in order. (The READ and WRITE rules may be interleaved.) The output values o_i come from applications of rule WRITE of Figure 3: its taint value is $tpc \vee tl$, for taint level tl of the written byte joined with the control flow taint level tpc.

Definition 1. *A* trace $tr(h_1, \ldots, h_n) \rightarrow o_1, \ldots, o_k$ *is* **no-taint** *when the taint level of every o_i is L. A* program *satisfies* **no-taint** *when every possible execution trace is* **no-taint**.

Definition 2 (Attacker model). *Our notion of security is derived from a game experiment. Assume the program is run with high inputs $h_1, \ldots, h_n$ chosen uniformly (i.e. randomly with uniform distribution) from the space of possible inputs M. Then, an attacker is given the outputs $o_1, \ldots, o_k$ of the execution trace $tr(h_1, \ldots, h_n) \rightarrow o_1, \ldots, o_k$, and is asked which inputs were used. We say the program is* **secure** *if the probability of guessing the inputs is $1/|M|$ for every trace.*

This notion of security is related to the classical definition non-interference [1]; for convenience, we took a probabilistic setting in which security is defined as a game since it simplifies the reasoning w.r.t. our byte-granulated memory and low-level machine representation.

Theorem 1. *If a program satisfies* **no-taint** *then it is* **secure**.

Note that its converse does not hold in general. For instance, consider the code of Figure 1 modified such that it writes "valid" instead of "invalid". In that case, all tainted outputs are the same and thus the code is actually safe, although **no-taint** is not passed. As such, this specific case must be handled manually in our current setting.

Region-Based Semantics. The C code of Figure 1 does not pass **no-taint**: after the for loop in is_valid(P), there is a H control flow taint *tpc*, and hence the output gets an H as well.

The first step into extending the above semantics to prevent overtainting is to construct a control flow graph. For simplicity, we assume there is a single function defined, so that there is only one sequence of basic blocks. LLVM's control flow graph is then a directed, connected graph that contains the different basic blocks of a program. It has a single start and end node, from which other nodes are reached. Control flow graphs for the examples in the paper are shown in Figure 5. In each case we can see the entry block and exit blocks.

The following are standard graph theory and compilers concepts [7]. A basic block bb_0 *dominates* basic block bb_1 when every path from the entry to bb_1 contains bb_0. Also, bb_1 *postdominates* bb_0 if every path from bb_0 to the exit passes trough bb_1. A *region* is a connected subgraph of the control flow graph that has exactly two connections to the remaining graph: an entry and an exit (this is why they are also known as single-entry-single-exit (SESE) regions). We can then characterize an SESE region by a pair of blocks: the entry and exit

blocks to the region. The *entry basic block* of a region is passed through when entering the region; it is considered part of the region, and dominates all basic blocks in the region. Similarly, the *exit basic block* is passed through after leaving the region; it is not considered part of the region.

Figure 5 shows the control graphs as computed by LLVM. In the control flow graph shown in Figure 5(a), corresponding to the code of Figure 3, there are several regions: a large region containing all basic blocks with *entry* and exit *bb*256 (which is *not* part of the region, and singleton blocks with entry *bbX* (for $X < 256$) and exit *bb*256. (In practice, LLVM static analysis ignores singleton regions, as they are always contained in a larger region.) In the control flow graph shown in Figure 5(b), corresponding to the code of Figure 4, there are two regions: one with entry *entry* and exit *out*, and one with entry *loop* and exit *out*.

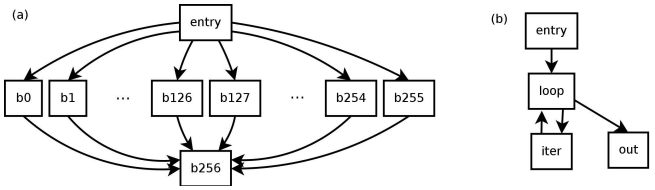

Fig. 5. Control flow graphs of basic blocks. (a) Graph for code of Figure 3, (b) Graph for code of Figure 4.

The important point is that control flow information (e.g., information that was added in a H branch to the control flow taint) has a region as its scope: the scope is pushed when entering a region (i.e., entering its entry basic block), and cleared (i.e., popped) once exiting the region (that is, we pass through its exit basic block), so we are free to reset the control flow taint to the previous state, since no information can be leaked anymore. In order to implement regions in the semantics, we need to add a stack of control flow taints. The rule is that whenever we enter a new region, we need to push a L value in the stack; whenever we exit, we pop the head of the stack. We use the following auxiliary function, that manipulates the stack:

$$region(pc, s, tl) = \begin{cases} push(s, head(s) \vee tl) & \\ & \text{if } pc = \text{entry into new region} \\ pop(s) & \text{if } pc = \text{exit current region} \\ s & \text{otherwise} \end{cases}$$

This function is built using a previous pass of the code through LLVM's static analysis to compute regions. Initially, stack s is empty. As we enter into new regions, we push new control flow taint levels.

The new method, called **precise-no-taint**, is similar to **no-taint** of the previous section but operates in the new semantics, and is thus more precise. We show this more precise semantics is as secure as the previous one.

Theorem 2. *If a program satisfies* **precise-no-taint** *then it is* **secure**.

The proof is similar to that of Theorem 1, although it now requires modularizing per region; H information that happened in a closed region in the past does not influence L outputs.

4 Conclusions

Tainting is an important technique to find dangerous information flows. Our addition of a dynamic tainting mechanism to KLEE is natural and complementary to the safety checks done by KLEE alone, like memory errors and overflows.

Our method can be used in isolation or coupled with previous work in order to verify code that uses cryptography; this framework is promising for analysing complex and long cryptographic protocol implementations. A longer version with more details, prototype code (as a patch to the latest KLEE distribution), and proof sketches can be found on our project website [4].

As future work, we intend to:

- Analyze larger, real-life code with subtle manipulation of sensitive data, in a similar vein to the toy example of Figure 1.
- Use the present analysis to prove secrecy of cryptographic materials like encryption keys; this will enable us to abstract away from concrete cryptographic primitives and use abstract, symbolic counterparts, as done in [5].
- Finally, we also we want to investigate more precise memory pointer tainting, as discussed in Section 2.

This work has been supported by the European Union Seventh Framework Programme under grant agreement no. 295261 (MEALS), and by the PICT-PRH 316 project.

References

1. Barthe, G., Rezk, T.: Non-interference for a jvm-like language. In: TLDI 2005, pp. 103–112. ACM, New York (2005)
2. Brumley, D., Caballero, J., Liang, Z., Newsome, J., Song, D.: Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In: USENIX, SS 2007, pp. 15:1–15:16. USENIX Association, Berkeley (2007)
3. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of OSDI, pp. 209–224 (2008)
4. Corin, R., Manzano, F.: Dynamic taint analysis for the klee symbolic execution engine (extended version), `http://cs.famaf.unc.edu.ar/~rcorin/kleecrypto`
5. Corin, R., Manzano, F.: Efficient Symbolic Execution for Analysing Cryptographic Protocol Implementations. In: Erlingsson, Ú., Wieringa, R., Zannone, N. (eds.) ESSoS 2011. LNCS, vol. 6542, pp. 58–72. Springer, Heidelberg (2011)

6. Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
7. Johnson, R., Pearson, D., Pingali, K.: The program structure tree: Computing control regions in linear time, pp. 171–185 (1994)
8. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
9. Lattner, C., Adve, V.: The LLVM language reference manual, `http://llvm.org/docs/LangRef.html`
10. Rizzo, J., Duong, T.: Practical padding oracle attacks. In: Proceedings of the 4th USENIX Conference on Offensive Technologies, WOOT 2010, pp. 1–8. USENIX Association, Berkeley (2010)
11. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE JSAC 21(1), 5–19 (2003)
12. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask) (2010)
13. Vaudenay, S.: Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS... In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 534–546. Springer, Heidelberg (2002)
14. Zhao, J., Nagarakatte, S., Martin, M.M.K., Zdancewic, S.: Formalizing the llvm intermediate representation for verified program transformations. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, pp. 427–440. ACM, New York (2012)