Static Analysis of String Manipulations in Critical Embedded C Programs

Xavier Allamigeon, Wenceslas Godard, and Charles Hymans

EADS CCR DCR/STI/C
12, rue Pasteur – BP 76 – 92152 Suresnes – France
`firstname.lastname@eads.net`

Abstract. This paper describes a new static analysis to show the absence of memory errors, especially string buffer overflows in C programs. The analysis is specifically designed for the subset of C that is found in critical embedded software. It is based on the theory of abstract interpretation and relies on an abstraction of stores that retains the length of string buffers. A *transport structure* allows to change the granularity of the abstraction and to concisely define several inherently complex abstract primitives such as destructive update and string copy. The analysis integrates several features of the C language such as multi-dimensional arrays, structures, pointers and function calls. A prototype implementation produces encouraging results in early experiments.

1 Introduction

Programming in C with strings, and more generally with buffers, is risky business. Before any copy, the programmer should make sure that the destination buffer is large enough to accept the source data in its entirety. When it is not the case, random bytes may end up in unexpected memory locations. This scenario is particularly unpleasant as soon as the source data can somehow be forged by an attacker: he may be able to smash [19] the return address on the stack and run its own code instead of the sequel of the program. Indeed, buffer overflows account for more than half of the vulnerabilities reported by the CERT [13] and are a popular target for viruses [10].

Needless to say defects that may abandon control of the equipment to an intruder are unacceptable in the context of embedded software. Testing being not a proof, we aim at designing a static analysis that shows the absence of memory manipulation errors (buffer, string buffer and pointer overflows) in embedded C software. We expect such a tool to be sound; to yield as few false alarms as possible in practice; to require as less human intervention as possible (manual annotations are unsuitable) and to scale to realistically sized programs. Any software engineer would easily benefit from a tool with all these traits and could rely on its results. Obviously the analysis should be able to handle all the features of the C language that are used in practice in the embedded world. This requires the smooth integration of several analysis techniques together. Simplicity of design is also a crucial point, since the analysis implementation should be bug-free and

cmd ::=		command	lv ::=		left value
$\mid$	$\tau\ x; cmd$	variable declaration	$\mid$	x	variable
$\mid$	$lv = e$	assignment	$\mid$	$lv.f$	field access
$\mid$	$?(lv \bowtie \text{'\textbackslash 0'})$	guard	$\mid$	$lv[e]$	array access
$\mid$	$cmd_1 + cmd_2$	alternative	$\mid$	$*e$	pointer deref.
$\mid$	$cmd_1; cmd_2$	sequence	e ::=		expression
$\mid$	cmd^*	arbitrary loop	$\mid$	c	constant
$\mid$	$f()$	function call	$\mid$	lv	left value
τ ::=		type	$\mid$	$e_1 \odot e_2$	binary operation
$\mid$	β	scalar type	$\mid$	$\&lv$	address of
$\mid$	$\tau[n]$	array type	$\mid$	$(\tau*)e$	cast
$\mid$	$\{\tau_1\ f_1 \dots \tau_n\ f_n\}$	structure type	$\odot$ ::=		binary operator
β ::=		scalar type	$\mid$	$+, -, \times, /$	integer arithmetic
$\mid$	**char**	character type	$\mid$	$+, -$	pointer arithmetic
$\mid$	**int**	integer type	$\bowtie$ ::=		comparison
$\mid$	$\tau*$	pointer type	$\mid$	$=$	equality
			$\mid$	$\neq$	difference

Fig. 1. Syntax

maintainable. This is not an easy task especially for a language as complex as C. To attain these goals we adopt the methodology of abstract interpretation [5]: section 2 presents the subset of C we tackle and its concrete semantics; section 3 describes the abstraction of strings and the sound static analysis algorithm; section 4 shows how string copy operations are handled, and what checks are performed by the tool; sections 5 and 6 address the implementation, experiments and related work.

2 Embedded C programs

2.1 Syntax

The C programming language is inherently complex, which makes the formal definition of its semantics a difficult task. Hopefully and for obvious safety reasons, programming critical embedded applications is subject to severe constraints. In practice, only a subset of C is allowed. The main limitation results from the obligation to know at compile time the maximum memory usage of any piece of software. To achieve this, the use of dynamic allocation (function *malloc()*) and recursive functions are both forbidden. For the sake of expositional clarity, we set aside some additional features such as numerous C scalar types, union types and goto statements. Dealing with these features brings issues orthogonal to the object of this paper. Some ideas to address these issues may be found in [7, 21]. In the end, we consider the relatively small kernel language with the syntax of figure 1. Complex assignments in C are broken down to simpler assignments between scalar types. All variables declared in a given scope of the program have distinct names.

2.2 Store

A memory address is a pair (x, o) of a variable identifier in $\mathcal{V}$ and an offset. It denotes the o^{th} byte from the address at which the content of variable x is stored. Operation $\boxplus$ shifts an address by a given offset:

$$(x, o) \boxplus i = (x, o + i)$$

Programs manipulate three kinds of basic values: integers in $\mathbb{Z}$, characters in $\mathbb{C}$ and pointers in $\mathbb{P}$. For sake of simplicity, integers are unbounded. The nature of characters is left unspecified. It is sufficient to say that there is one null character denoted by '\0'. A pointer is a triple $\langle a, i, n\rangle$ that references the i^{th} byte of a buffer that starts from address a and is n bytes long.

The store maps each allocated address to a basic value. The kind of values stored at a given address never changes. Hence, a store $\sigma = (\sigma_{\mathbb{Z}}, \sigma_{\mathbb{C}}, \sigma_{\mathbb{P}})$ defined on allocated addresses $\mathcal{A} = \mathcal{A}_{\mathbb{Z}} \oplus \mathcal{A}_{\mathbb{C}} \oplus \mathcal{A}_{\mathbb{P}}$ belongs to the set:

$$\Sigma = (\mathcal{A}_{\mathbb{Z}} \to \mathbb{Z}) \times (\mathcal{A}_{\mathbb{C}} \to \mathbb{C}) \times (\mathcal{A}_{\mathbb{P}} \to \mathbb{P})$$

In this model, any operation that alters the interpretation of data too severely leads to an error at runtime. For instance, a cast from $\{\textbf{int}\ a;\ \textbf{int}\ b\}*$ to $\textbf{int}*$ is valid; whereas a cast from $\textbf{int}*$ to $\textbf{char}*$ is illegitimate.

The layout of memory is given by two functions: $\mathsf{sizeof}(\tau)$ returns the size of a data of type τ and $\mathsf{offset}(f)$ the offset of a field f from the beginning of its enclosing structure.

2.3 Semantics

We assign a denotational semantics [29] to the kernel language. In the following, we use notations $lv : \tau$ and $e : \tau$ to retrieve the type τ of a left value lv or expression e as computed by a standard C typechecker. A left value lv evaluates to a set of addresses $\mathcal{L}\{\!|lv|\!\}$, as formalized in figure 2. Sets allow to encode both non-determinism and halting behaviours. A pointer of type $\tau*$ can be safely dereferenced, as long as there remains enough space to store an element of type τ. Likewise, an expression e of integer type evaluates to a set of integers $\mathcal{R}_{\mathbb{Z}}\{\!|e|\!\}$. Notice how an access to some address not allocated in the store of integer halts program execution. We skip the classical definition of relation $v_1 \odot v_2 \Rightarrow v$ which explicits the meaning of each binary operation. Definitions for expressions of character or pointer type are completely identical. The last four equations in figure 2 define pointer creation and cast.

$$
\begin{aligned}
\mathcal{L}\{\!|x|\!\}\sigma &= \{(x,0)\}\\
\mathcal{L}\{\!|lv.f|\!\}\sigma &= \{a \boxplus \mathsf{offset}(f) \mid a \in \mathcal{L}\{\!|lv|\!\}\sigma\}\\
\mathcal{L}\{\!|lv[e]|\!\}\sigma &= \{a \boxplus i \times \mathsf{sizeof}(\tau) \mid a \in \mathcal{L}\{\!|lv|\!\}\sigma \wedge i \in \mathcal{R}_{\mathbb{Z}}\{\!|e|\!\}\sigma \wedge 0 \le i < n\}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{where } lv : \tau[n]\\
\mathcal{L}\{\!|{*e}|\!\}\sigma &= \{a \boxplus i \mid \langle a, i, n\rangle \in \mathcal{R}_{\mathbb{P}}\{\!|e|\!\}\sigma \wedge 0 \le i \le n - \mathsf{sizeof}(\tau)\} \quad \text{where } e : \tau*\\
\mathcal{R}_{\mathbb{Z}}\{\!|c|\!\}\sigma &= \{c\}\\
\mathcal{R}_{\mathbb{Z}}\{\!|lv|\!\}\sigma &= \{\sigma_{\mathbb{Z}}(a) \mid a \in \mathcal{L}\{\!|lv|\!\}\sigma \cap \mathsf{dom}(\sigma_{\mathbb{Z}})\}\\
\mathcal{R}_{\mathbb{Z}}\{\!|e_1 \odot e_2|\!\}\sigma &= \{v \mid v_1 \in \mathcal{R}_{\mathbb{Z}}\{\!|e_1|\!\}\sigma \wedge v_2 \in \mathcal{R}_{\mathbb{Z}}\{\!|e_2|\!\}\sigma \wedge v_1 \odot v_2 \Rightarrow v\}\\
\mathcal{R}_{\mathbb{P}}\{\!|\&x|\!\}\sigma &= \{\langle (x,0), 0, \mathsf{sizeof}(\tau)\rangle\} \qquad\qquad\qquad\qquad \text{where } x : \tau\\
\mathcal{R}_{\mathbb{P}}\{\!|\&lv.f|\!\}\sigma &= \{\langle a, 0, \mathsf{sizeof}(\tau)\rangle \mid a \in \mathcal{L}\{\!|lv.f|\!\}\sigma\} \qquad\qquad \text{where } lv.f : \tau\\
\mathcal{R}_{\mathbb{P}}\{\!|\&lv[e]|\!\}\sigma &= \{\langle a, i \times \mathsf{sizeof}(\tau), \mathsf{sizeof}(\tau[n])\rangle \mid a \in \mathcal{L}\{\!|lv|\!\}\sigma \wedge i \in \mathcal{R}_{\mathbb{Z}}\{\!|e|\!\}\sigma\}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{where } lv : \tau[n]\\
\mathcal{R}_{\mathbb{P}}\{\!|(\tau*)e|\!\}\sigma &= \mathcal{R}_{\mathbb{P}}\{\!|e|\!\}\sigma
\end{aligned}
$$

Fig. 2. Semantics of left values and expressions

Three atomic commands operate on the store:

$$
\begin{aligned}
\mathcal{C}\{\!|\tau\ x; cmd|\!\}\sigma &= \{\sigma'_{|\mathsf{dom}(\sigma)} \mid \sigma_x = \mathcal{I}\{\!|\tau|\!\}(x,0) \wedge \sigma' \in \mathcal{C}\{\!|cmd|\!\}(\sigma \oplus \sigma_x)\}\\
&\qquad\qquad\qquad\qquad \text{where } \mathsf{dom}(\sigma) \cap \mathsf{dom}(\sigma_x) = \emptyset\\
\mathcal{C}\{\!|lv = e|\!\}\sigma &= \{\sigma[a \mapsto v] \mid a \in \mathcal{L}\{\!|lv|\!\}\sigma \cap \mathsf{dom}(\sigma_{\mathbb{Z}}) \wedge v \in \mathcal{R}_{\mathbb{Z}}\{\!|e|\!\}\sigma\}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{where } lv, e : \mathbb{Z}\\
\mathcal{C}\{\!|?(lv \bowtie \text{'\textbackslash 0'})|\!\}\sigma &= \{\sigma \mid v \in \mathcal{R}_{\mathbb{C}}\{\!|lv|\!\}\sigma \wedge v \bowtie \text{'\textbackslash 0'}\}
\end{aligned}
$$

At variable declaration, a new store fragment σ_x is initialized and concatenated to the existing store, execution then continues until variable x is eventually deleted from the resulting store. The new store fragment is built by induction on the type of the declared variable:

$$
\begin{aligned}
\mathcal{I}\{\!|\beta|\!\}a &= [a \mapsto v]\\
\mathcal{I}\{\!|\tau[n]|\!\}a &= \bigoplus_{0 \le i < n} \mathcal{I}\{\!|\tau|\!\}(a \boxplus i \times \mathsf{sizeof}(\tau))\\
\mathcal{I}\{\!|\{\tau_1 f_1 \ldots \tau_n f_n\}|\!\}a &= \bigoplus_{0 < i \le n} \mathcal{I}\{\!|\tau_i|\!\}(a \boxplus \mathsf{offset}(f_i))
\end{aligned}
$$

where v is any value of type β and $\oplus$ joins two disjoint stores. Assignments come in three flavours, one for each basic type: integer, character and pointer. Here, we only describe the integer assignment since the other two are completely similar. Assignment to a non-allocated address brings the program to a halt. Guards let execution continue when the store satisfies the boolean condition. We consider only equality or disequality with the null character even though other kinds of

$$
\begin{array}{lll}
\mathbb{C}^\sharp = \{\bot_{\mathbb{C}}, \mathbf{0}, \mathbf{1}, \top_{\mathbb{C}}\} & \mathbb{Z}^\sharp = (\overline{\mathbb{Z}} \times \overline{\mathbb{Z}})_\bot & \mathbb{A}^\sharp = (\mathcal{V} \times \mathbb{Z}^\sharp)_\bot^\top \\
\gamma_{\mathbb{C}}(\bot_{\mathbb{C}}) = \emptyset \quad \gamma_{\mathbb{C}}(\top_{\mathbb{C}}) = \mathbb{C} & \gamma_{\mathbb{Z}}(\bot_{\mathbb{Z}}) = \emptyset & \gamma_{\mathbb{A}}(\bot_{\mathbb{A}}) = \emptyset \\
\gamma_{\mathbb{C}}(\mathbf{0}) = \{\text{'\textbackslash 0'}\} & \gamma_{\mathbb{Z}}([l;u]) = \{i \mid l \leq i \leq u\} & \gamma_{\mathbb{A}}(\top_{\mathbb{A}}) = \mathbb{A} \\
\gamma_{\mathbb{C}}(\mathbf{1}) = \mathbb{C} \setminus \{\text{'\textbackslash 0'}\} & & \gamma_{\mathbb{A}}(x, O) = \{(x, o) \mid o \in \gamma_{\mathbb{Z}}(O)\}
\end{array}
$$

$$
\begin{aligned}
\mathbb{P}^\sharp &= (\mathbb{A}^\sharp \times \mathbb{Z}^\sharp \times \mathbb{Z}^\sharp)_\bot \\
\gamma_{\mathbb{P}}(\bot_{\mathbb{P}}) &= \emptyset \\
\gamma_{\mathbb{P}}(A, I, N) &= \{\langle a, i, n\rangle \mid a \in \gamma_{\mathbb{A}}(A) \wedge i \in \gamma_{\mathbb{Z}}(I) \wedge n \in \gamma_{\mathbb{Z}}(N)\}
\end{aligned}
$$

Fig. 3. Abstract addresses and values

guards may easily be handled. The remaining commands control the flow of execution:

$$
\begin{aligned}
\mathcal{C}\{\!|cmd_1 + cmd_2|\!\}\sigma &= \mathcal{C}\{\!|cmd_1|\!\}\sigma \cup \mathcal{C}\{\!|cmd_2|\!\}\sigma \\
\mathcal{C}\{\!|cmd_1; cmd_2|\!\}\sigma &= \mathcal{C}\{\!|cmd_2|\!\}(\mathcal{C}\{\!|cmd_1|\!\}\sigma) \\
\mathcal{C}\{\!|cmd^*|\!\}\sigma &= \mathsf{lfp}_\emptyset\, \mathbb{F}_\sigma \\
\mathbb{F}_{\sigma_0}(X) &= \{\sigma_0\} \cup \{\sigma' \mid \sigma \in X \wedge \sigma' \in \mathcal{C}\{\!|cmd|\!\}\sigma\} \\
\mathcal{C}\{\!|f()|\!\}\sigma &= \mathcal{C}\{\!|cmd|\!\}\sigma \qquad \text{where } cmd \text{ is the body of function } f
\end{aligned}
$$

A program P consists of a set of functions and a main command which is executed in an initially empty store: $\{\!|P|\!\} = \mathcal{C}\{\!|cmd|\!\}\varepsilon$.

3 Static analysis

We wish to automatically verify that all string manipulations in a program are innocuous. This is, by nature, an undecidable problem. So, we design a static analysis that computes an approximate but sound representation of all the stores that result from the execution of a program. Following the methodology of abstract interpretation [5], an abstraction of sets of stores is first devised. The analysis algorithm is then systematically derived thanks to this abstraction from the concrete semantics. The results of the analysis are used to check as many potentially dangerous memory operations as possible and to emit warnings in other cases.

3.1 Abstract values, integer and pointer stores

Figure 3 lists the abstract domains and concretization functions used for sets of addresses and values. These abstractions are all built from well-known standard domains: integers are represented by ranges [5]; characters thanks to the domain of equality/disequality with the null character; a pair of a variable identifier and a range of possible offsets stands for a set of addresses; and abstract pointers are triples made of an abstract address, followed by two ranges for possible offsets and sizes. We use the standard set notations for all operations on ranges: $(\subseteq, \cap, \min, \max)$. Moreover, $I_1 \curlyvee I_2$ denotes the smallest range that contains both I_1 and I_2; $I \setminus \{n\}$ the smallest range that contains all elements in I except n; $I + n$ $(I - n)$ is the range obtained after the addition (subtraction) of n to all the elements in I.

The abstract domain $(\boldsymbol{D}, \boldsymbol{\gamma})$ of the analysis is built as the product of three domains: one for each type of basic value. An abstract store $\boldsymbol{S}$ is thus a triple $(S_{\mathbb{Z}}, S_{\mathbb{P}}, S_{\mathbb{C}})$. Abstract integer $S_{\mathbb{Z}}$ and pointer $S_{\mathbb{P}}$ stores map each allocated address to an abstract value of corresponding type. A fully fledged description of these standard non-relational domains is skipped. On the other hand, the abstract character store $S_{\mathbb{C}}$, being the object of our study, is discussed at length in the next section.

3.2 Abstract character store

A string in C is a sequence of characters stored in memory. The first null character ('\0') signals the end of the string. If no null character is found before the end of the allocated area, then the string is not well-formed. Hence the length of a string stored on a buffer $(a : n)$ of n consecutive bytes starting at address a in a store σ is:

$$\mathit{strlen}_{\sigma}(a : n) = \min(\{n\} \cup \{l \mid 0 \leq l < n \wedge \sigma(a \boxplus l) = \text{'\textbackslash 0'}\})$$

Now, in order to prove the correctness of string manipulations it is necessary to at least retain some information about the length of the various strings in the store.

Let π be a partition of the set of all allocated addresses, such that each element in the partition is a connected set (a buffer). The abstract store maps each buffer in the partition to a range that approximates the possible lengths of the string stored on that buffer:

$$\begin{aligned} \Sigma^{\sharp} &= (\pi \to \mathbb{Z}^{\sharp})_{\perp} \\ \gamma(S) &= \{\sigma \mid \forall b \in \pi : \mathit{strlen}_{\sigma}(b) \in \gamma_{\mathbb{Z}}(S(b))\} \\ \gamma(\perp) &= \emptyset \end{aligned}$$

Several primitives operate on the domain of character store. Each primitive obeys a soundness condition. Normalization returns the empty store as soon as any buffer is associated with an empty range:

$$\eta(S) = \begin{cases} \perp & \text{if } \exists b : S(b) = \perp_{\mathbb{Z}} \\ S & \text{otherwise} \end{cases}$$

Normalization preserves the meaning of the abstract store, thus: $\gamma(\eta(S)) = \gamma(S)$. From now on, we assume that the store is always in normal form so that no abstract length can ever be the empty range. A new abstract store with no information at all may be created using primitive $\mathsf{universe}$ from a partition π. It is such that for any buffer $(a : n)$ in π:

$$\mathsf{universe}(\pi)(a : n) = [0; n]$$

It is straightforward to show that: $(\mathcal{A} \to \mathbb{C}) \subseteq \gamma(\mathsf{universe}(\pi))$. Abstract stores S_1 and S_2 defined on the same partition π can be compared:

$$\begin{aligned} S_1 \sqsubseteq S_2 &\iff (S_1 = \bot \vee (S_2 \neq \bot \wedge \forall b \in \pi : S_1(b) \subseteq S_2(b))) \\ &\iff \gamma(S_1) \subseteq \gamma(S_2) \end{aligned}$$

Abstract join $\sqcup$ and meet $\sqcap$ operations are performed pointwise. To deal with variable declarations, we need to concatenate stores of disjoint domains and remove all the buffers allocated for a given variable:

$$\begin{aligned} \bot \oplus S = S \oplus \bot &= \bot \\ S_1 \oplus S_2 &= S_1 S_2 \\ S \setminus x &= S_{|\{(a:n)\in\pi \mid a=(y,o)\wedge y\neq x\}} \end{aligned}$$

These operations verify the following set inequalities:

$$\begin{aligned} \gamma(S_1) \cup \gamma(S_2) &\subseteq \gamma(S_1 \sqcup S_2) \\ \gamma(S_1) \cap \gamma(S_2) &\subseteq \gamma(S_1 \sqcap S_2) \\ \{\sigma_1\sigma_2 \mid \sigma_1 \in \gamma(S_1) \wedge \sigma_2 \in \gamma(S_2)\} &\subseteq \gamma(S_1 \oplus S_2) \\ \{\sigma_{|\{(y,o)\in\mathcal{A} \mid y\neq x\}} \mid \sigma \in \gamma(S)\} &\subseteq \gamma(S \setminus x) \end{aligned}$$

Boolean conditions present in if statements, switches and loops must be taken into account in order to produce sufficiently precise results. Primitive guard constrains the store according to an equality or disequality comparison with character '\0':

$$\{\sigma \mid \sigma \in \gamma(S) \wedge a \in \gamma(A) \cap \mathcal{A} \wedge \sigma(a) \bowtie \text{'\textbackslash 0'}\} \subseteq \gamma(\mathsf{guard}(A \bowtie \text{'\textbackslash 0'}, S))$$

Suppose the constraint implies that there is at least one '\0' character in a memory region that spans from address (x, o_1) to address (x, o_2). Suppose further that this region is contained in a unique buffer $(a : n)$ of the partition. Then, the length of a string starting in a is necessarily smaller than the distance δ from a to (x, o_2). Hence:

$$\mathsf{guard}((x, [o_1; o_2]) = \text{'\textbackslash 0'}, S) = \eta(S[a : n \mapsto S(a : n) \cap [0; \delta]])$$

Similarly, suppose now that the value stored at address (x, o) is not the '\0' character. If address (x, o) belongs to some buffer $a : n$ of the partition and δ is the distance from a to (x, o), then:

$$\mathsf{guard}((x, [o; o]) \neq \text{'\textbackslash 0'}, S) = \eta(S[b \mapsto S(b) \setminus \{\delta\}])$$

In all other cases, guard simply leaves the store unchanged:

$$\mathsf{guard}(A \bowtie \text{'\textbackslash 0'}, S) = S$$

Transport structure and store accesses. Operations to read and write in the store are primordial to the analysis. However they are not easily defined mainly because the region in memory that is impacted by the operation does not necessarily coincide with a particular buffer in the partition. In order to alleviate this difficulty, we first devise transformations on the abstract store that allow to change the underlying partition. Transformation cut C_δ splits the buffer b into two consecutive buffers b_1 and b_2 of respective sizes δ and n; the reverse transformation glue G_δ lumps together two buffers that are *contiguous*:

$$\mathsf{C}_\delta([b \mapsto L]) = \begin{cases} [b_1 \mapsto [\delta;\delta]; b_2 \mapsto L - \delta] & \text{if } \delta \leq \min(L) \\ [b_1 \mapsto L \cap [0;\delta]; b_2 \mapsto [0;n]] & \text{otherwise} \end{cases}$$

$$\mathsf{G}_\delta([b_1 \mapsto K; b_2 \mapsto L]) = \begin{cases} [b \mapsto K \curlyvee (L + \delta)] & \text{if } \delta \in K \\ [b \mapsto K] & \text{otherwise} \end{cases}$$

Both operations are sound in that their result includes at least all the concrete stores originally present:

$$\gamma(S \oplus [b \mapsto L]) \subseteq \gamma(S \oplus \mathsf{C}_\delta([b \mapsto L]))$$
$$\gamma(S \oplus [b_1 \mapsto K; b_2 \mapsto L]) \subseteq \gamma(S \oplus \mathsf{G}_\delta([b_1 \mapsto K; b_2 \mapsto L]))$$

Building on glue and cut, there is a simple algorithm to move from any partition π_1 to another π_2 (of course, π_1 and π_2 must be defined on the same set of allocated addresses). Starting from π_1, the first step consists in splitting buffers until we get to the coarsest partition which is finer than both π_1 and π_2. Then, in a second step buffers are glued together to get back to π_2. Let us introduce two very useful shortcut notations built on top of this algorithm. In the following, all addresses in buffer $b = (a : n)$ are allocated (i.e. $b \subseteq \mathcal{A}$):

– $\Phi S(b)$ minimally modifies the store so as to include buffer b in the resulting partition and then returns the value associated with this buffer. More accurately, let $\pi \oplus \{(a_1 : n_1) \ldots (a_k : n_k)\}$ be the initial partition, where all buffers that overlap b are listed in increasing order as $(a_1 : n_1)$ to $(a_k : n_k)$. Then the destination partition is $\pi \oplus \{(a_1 : \delta), (a : n), (a \boxplus n : \delta')\}$, where δ and δ' are the respective distances from a_1 to a and from $a \boxplus n$ to $a_k \boxplus n_k$,
– $\Phi S[b \mapsto L]$ transforms the partition to add buffer b as previously explained, updates its value with L and translates back to the initial partition.

Memory accesses can now be described by the equations of figure 4. Let us comment the cases when the abstract address A that is read or written is of the form $(x, [o_1; o_2])$ and all the addresses from (x, o_1) to (x, o_2) are allocated. In this case, the buffer b that corresponds to A starts in (x, o_1) and stretches over $n = (o_2 - o_1 + 1)$ bytes. Thanks to the previously introduced transformations, we can easily convert the abstract store so that buffer b belongs to the partition. Then, to evaluate the value that is read, we apply function eval_n to the abstract length L associated with b. There are three cases:

$$\mathsf{read}(S, A) = \begin{cases} \bot_{\mathbb{C}} & \text{if } S = \bot \vee A = \bot_{\mathbb{A}} \\ \mathsf{eval}_{|b|}(\Phi S(b)) & \text{if } A = (x, O) \wedge b = \mathsf{tobuff}(A) \wedge b \subseteq \mathcal{A} \\ \top_{\mathbb{C}} & \text{otherwise} \end{cases}$$

$$\mathsf{write}(S, A, V) =$$
$$\begin{cases} \bot & \text{if } S = \bot \vee A = \bot_{\mathbb{A}} \vee V = \bot_{\mathbb{Z}} \\ \Phi S[b \mapsto \mathsf{update}_{|b|}(S(b), V)] & \text{if } A = (x, O) \wedge b = \mathsf{tobuff}(A) \wedge b \subseteq \mathcal{A} \\ \mathsf{universe}(\pi) & \text{otherwise} \end{cases}$$

$$\mathsf{tobuff}(x, [o_1; o_2]) = ((x, o_1) : (o_2 - o_1 + 1))$$

$$\mathsf{eval}_n(L) = \begin{cases} \mathbf{0} & \text{if } n = 1 \wedge L = [0; 0] \\ \mathbf{1} & \text{if } L = [n; n] \\ \top_{\mathbb{C}} & \text{otherwise} \end{cases}$$

$$\mathsf{update}_n([l; u], \mathbf{0}) = [0; \min(u, n-1)]$$
$$\mathsf{update}_n([l; u], \mathbf{1}) = \begin{cases} [1; 1] & \text{if } n = 1 \\ [l; n] & \text{otherwise} \end{cases}$$
$$\mathsf{update}_n(L, \top) = [0; n]$$

Fig. 4. Abstract memory access

- when the buffer contains only one character that is equal to '\0', then **0** is returned,
- when $L = [n; n]$, the first '\0' character is not in the buffer, so the returned value is **1**,
- in all other cases, there is insufficient information to conclude and $\top_{\mathbb{C}}$ is returned.

The intuition that motivates definition of function update goes as follows:

- After a '\0' character is written somewhere in the buffer, we can be sure that the length is strictly less than its size n. Moreover, previous '\0' characters remain so that $\mathsf{update}_n([l; u], \mathbf{0}) = [0; \min(u, n-1)]$.
- If exactly one non-null character is copied in a buffer of size $n = 1$, then the first '\0' can not be at index 0, so $\mathsf{update}_1(L, \mathbf{0}) = [1; 1]$.
- In the remaining cases when a non-null character is written, it may erase the first '\0' character in the buffer, so that the length of the string may be unbounded. Since non-null characters are untouched, the information about the lower bound on the possible string lengths is kept, thus $\mathsf{update}_n([l; u], \mathbf{0}) = [l; n]$.
- At last, when an unknown value is copied, all information is lost.

These operations are sound with respect to:

$$\{\sigma(a) \mid \sigma \in \gamma(S) \wedge a \in \gamma_{\mathbb{A}}(A) \cap \mathsf{dom}(\sigma)\} \subseteq \gamma_{\mathbb{C}}(\mathsf{read}(S, A))$$
$$\{\sigma[a \mapsto v] \mid \sigma \in \gamma(S) \wedge a \in \gamma_{\mathbb{A}}(A) \cap \mathsf{dom}(\sigma) \wedge v \in \gamma_{\mathbb{C}}(V)\} \subseteq \gamma(\mathsf{write}(S, A, V))$$

$$\mathcal{R}_{\mathbb{C}}[\![c]\!]\boldsymbol{S} = \begin{cases}\mathbf{0} & \text{if } c = \text{'\textbackslash 0'}\\ \mathbf{1} & \text{otherwise}\end{cases}$$

$$\mathcal{R}_{\mathbb{C}}[\![lv]\!]\boldsymbol{S} = \mathsf{read}(\boldsymbol{S}_{\mathbb{C}}, \mathcal{L}[\![lv]\!]\boldsymbol{S})$$

$$\begin{aligned}
\mathcal{I}[\![\mathbf{char}]\!]a &= (\bot, \bot, \mathsf{universe}(\{(a,1)\}))\\
\mathcal{I}[\![\mathbf{char}[n]]\!]a &= (\bot, \bot, \mathsf{universe}(\{(a,n)\}))\\
\mathcal{I}[\![\tau[n]]\!]a &= \bigoplus_{0\le i<n} \{\mathcal{I}[\![\tau]\!](a \boxplus i \times \mathsf{sizeof}(\tau))\}\\
\mathcal{I}[\![\{\tau_1 f_1 \ldots \tau_n f_n\}]\!]a &= \bigoplus_{0<i\le n} \{\mathcal{I}[\![\tau_i]\!](a \boxplus \mathsf{offset}(f_i))\}
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}[\![\tau\ x; cmd]\!]\boldsymbol{S} &= \mathcal{C}[\![cmd]\!](S \oplus \mathcal{I}[\![\tau]\!](x,0)) \setminus x\\
\mathcal{C}[\![lv = e]\!](S_{\mathbb{Z}}, S_{\mathbb{P}}, S_{\mathbb{C}}) &= (S_{\mathbb{Z}}, S_{\mathbb{P}}, \mathsf{write}(S_{\mathbb{C}}, \mathcal{L}[\![lv]\!]S, \mathcal{R}_{\mathbb{C}}[\![e]\!])) \qquad \text{where } lv, e : \mathbf{char}\\
\mathcal{C}[\![?(lv \bowtie \text{'\textbackslash 0'})]\!](S_{\mathbb{Z}}, S_{\mathbb{P}}, S_{\mathbb{C}}) &= (S_{\mathbb{Z}}, S_{\mathbb{P}}, \mathsf{guard}(\mathcal{L}[\![lv]\!]S \bowtie \text{'\textbackslash 0'}, S_{\mathbb{C}}))\\
\mathcal{C}[\![cmd_1 + cmd_2]\!]\boldsymbol{S} &= \mathcal{C}[\![cmd_1]\!]\boldsymbol{S} \sqcup \mathcal{C}[\![cmd_2]\!]\boldsymbol{S}\\
\mathcal{C}[\![cmd_1; cmd_2]\!]\boldsymbol{S} &= \mathcal{C}[\![cmd_2]\!](\mathcal{C}[\![cmd_1]\!]\boldsymbol{S})\\
\mathcal{C}[\![cmd^*]\!]\boldsymbol{S} &= \mathsf{lfp}_{\bot}\ \mathbb{F}^{\sharp}_{\boldsymbol{S}}\\
\mathbb{F}^{\sharp}_{\boldsymbol{S_0}}(\boldsymbol{S}) &= S_0 \sqcup \mathcal{C}[\![cmd]\!]\boldsymbol{S}\\
\mathcal{C}[\![f()]\!]\boldsymbol{S} &= \mathcal{C}[\![cmd]\!]\boldsymbol{S} \qquad \text{where } cmd \text{ is the body of function } f
\end{aligned}$$

Fig. 5. Abstract evaluation, initialization and execution of commands

3.3 Abstract semantics

Building on the previous primitives, the static analysis computes abstract stores while mimicking the concrete semantics. Figure 5 presents the definition that are specifically related to the handling of characters and strings. The remaining aspects of the analysis are standard and thus not thoroughly described here.

Let us paraphrase some of the most spicy equations:

- To initialize a zone of memory starting at address a with a single character or with an array of n characters, $\mathcal{I}\{\![\tau]\!\}a$ creates a store whose partition is reduced to a unique buffer of size 1 or n and that contains no information,
- Non-deterministic choice amounts to abstract join and the sequence to function composition,
- The abstract store after a loop is the result of an abstract fixpoint computation. The constructive version of Tarski's theorem [6] suggests a naive algorithm: starting from $\bot$, the successive iterates of $\mathbb{F}^{\sharp}$ are computed until stabilization. In practice other more complex algorithms [31], the use of widening, and loop unfolding may be safely applied.

Theorem 1 (Soundness). *The abstract semantics of a command cmd on an abstract store $\boldsymbol{S}$ includes all stores that are obtained by any run of the command starting from some initial store in $\boldsymbol{\gamma}(\boldsymbol{S})$:*

$$\{\sigma' \mid \sigma \in \boldsymbol{\gamma}(\boldsymbol{S}) \wedge \sigma' \in \mathcal{C}\{\![cmd]\!\}\sigma\} \subseteq \boldsymbol{\gamma}(\mathcal{C}[\![cmd]\!]\boldsymbol{S})$$

Proof. The proof is done by structural induction on the syntax of commands. It reduces to the assembly of the various atomic soundness conditions of each primitive.

Note, that since, our static analysis is built in a modular way, it would be possible to replace some components to improve either precision or efficiency and still retain the overall soundness theorem. In particular any other non-relational numerical domain can be easily used instead of ranges.

Example 1. Here are the invariants collected by the static analysis with the character store for a small example:

`l0: char buf[10];`	$(\texttt{buf}, 0) : 10 \mapsto [0; 10]$
`l1: buf[0] = 'a';`	$(\texttt{buf}, 0) : 10 \mapsto [1; 10]$
`l2: buf[4] = '\0';`	$(\texttt{buf}, 0) : 10 \mapsto [1; 4]$
`l3: buf[1] = 'b';`	$(\texttt{buf}, 0) : 10 \mapsto [2; 10]$
`l4: buf[2] = '\0';`	$(\texttt{buf}, 0) : 10 \mapsto [2; 2]$

The partition is reduced to one buffer that starts at $(\texttt{buf}, 0)$ of 10 bytes. Let us delve into the details of the computation from label `l2` to `l3`. The tool reaches label `l2` with the knowledge that the length of `buf` is greater than 1:

$$(\texttt{buf}, 0) : 10 \mapsto [1; 10]$$

The partition is split in three around the zone that is being written:

$$\begin{gathered}(\texttt{buf}, 0) : 4 \mapsto [1; 4]\\(\texttt{buf}, 4) : 1 \mapsto [0; 1]\\(\texttt{buf}, 5) : 5 \mapsto [0; 5]\end{gathered}$$

The null character is written in buffer $(\texttt{buf}, 4) : 1$, using primitive update:

$$\begin{gathered}(\texttt{buf}, 0) : 4 \mapsto [1; 4]\\(\texttt{buf}, 4) : 1 \mapsto [0; 0]\\(\texttt{buf}, 5) : 5 \mapsto [0; 5]\end{gathered}$$

At last, the buffers are glued together to restore the initial partition:

$$(\texttt{buf}, 0) : 10 \mapsto [1; 4]$$

Note that at instruction `l3`, after character `'b'` is written at index `1` of `buf`, the upper bound on the length of the string is forgotten. This is indeed necessary. Consider the concrete store where the first '\0' character is exactly at index `1`; since it is overwritten by a non-null character and the tool has no information about the position of the remaining '\0' characters after the first one, the new length is unknown.

Imagine now that the previous example were ended by a call to `strcpy` that copies string `buf` into a buffer of size strictly larger than 2. Such a call would be correct and the approximation computed by the tool precise enough to prove this. Next section is about the analysis of the `strcpy` and the checks that are made to show the correctness of possibly dangerous memory manipulation operations.

$$\mathsf{strlen}(S, A) = \begin{cases} \bot_{\mathbb{Z}} & \text{if } S = \bot \vee A = \bot_{\mathbb{A}} \\ \Phi S((x,o) : m) \setminus \{m\} & A = (x, [o; o]) \wedge m = \mathsf{allocsz}_{\mathcal{A}}(x, o) \\ \mathsf{weakstrlen}(S, b) & \text{if } A = (x, O) \wedge b = \mathsf{tobuff}(A) \\ [0; +\infty] & \text{otherwise} \end{cases}$$

$$\mathsf{strcpy}(S, A, \bot) = \bot$$

$$\mathsf{strcpy}(S, A, [l; u]) =$$

$$\begin{cases} \bot & \text{if } S = \bot \vee A = \bot_{\mathbb{A}} \\ \eta(\Phi S[(x,o) : m \mapsto [l; m-1]]) & \text{if } A = (x, [o; o]) \wedge m = \min(u+1, \mathsf{allocsz}_{\mathcal{A}}(x, o)) \\ \eta(\mathsf{weakstrcpy}(S, b, [l; u])) & \text{if } A = (x, O) \wedge b = \mathsf{tobuff}(A) \wedge b \subseteq \mathcal{A} \\ \mathsf{universe}(\pi) & \text{otherwise} \end{cases}$$

$$\mathsf{weakstrlen}(S, X) = \bigvee \{\Phi S(a : m) \setminus \{m\} \mid a \in X \wedge m = \mathsf{allocsz}_{\mathcal{A}}(a)\}$$

$$\mathsf{weakstrcpy}(S, a : n, [l; u]) = \Phi S[a : m \mapsto \Phi S(a : m) \curlyvee [l; m-1]] \quad \text{where } m = \min(u + n, \mathsf{allocsz}_{\mathcal{A}}(a))$$

Fig. 6. Abstract string length and string copy

4 String copy

4.1 Concrete semantics

The syntax of commands is enriched with $\texttt{strcpy}(e_1, e_2)$. This call copies the string pointed to by pointer e_2 into the buffer starting in e_1:

$$\mathcal{C}\{\!|\texttt{strcpy}(e_1, e_2)|\!\}\sigma = \left\{ \sigma[a_1 \boxplus j \mapsto \sigma(a_2 \boxplus j)]_{0 \le j \le l} \;\middle|\; \begin{array}{l} a_1 \in \mathcal{L}\{\!|{*e_1}|\!\}\sigma \wedge n_1 = \mathsf{allocsz}_{\mathcal{A}}(a_1) \\ a_2 \in \mathcal{L}\{\!|{*e_2}|\!\}\sigma \wedge n_2 = \mathsf{allocsz}_{\mathcal{A}}(a_2) \\ l = \mathit{strlen}_\sigma(a_2 : n_2) \wedge l \neq n_2 \wedge l < n_1 \end{array} \right\}$$

In the previous equation $\mathsf{allocsz}(a)$ denotes the number of bytes that are allocated starting from address a. The source buffer denoted by e_2 should contain a valid string, i.e. there should be some '\0' character before the end of the allocated source memory zone. In other words, the length $l = \mathit{strlen}_\sigma(a_2 : n_2)$ of the string should be different from n_2. Additionally, l should be smaller than the size n_1 of the destination buffer. Otherwise, there is not enough space to copy the entire string and, according to this semantics, the program halts.

4.2 Abstract semantics

In the abstract world, `strcpy` is performed in two phases:

$$\mathcal{C}[\![\texttt{strcpy}(e_1, e_2)]\!]\boldsymbol{S} = (\boldsymbol{S}_{\mathbb{Z}}, \boldsymbol{S}_{\mathbb{P}}, \mathsf{strcpy}(\boldsymbol{S}_{\mathbb{C}}, \mathcal{L}[\![{*e_1}]\!]\boldsymbol{S}, \mathsf{strlen}(\boldsymbol{S}_{\mathbb{C}}, \mathcal{L}[\![{*e_2}]\!]\boldsymbol{S})))$$

Both phases are defined in figure 6. First, strlen retrieves the length of the source string. When the address a of the source string is exactly known, it reads the information associated with the buffer that starts from a and goes until the first non-allocated address $a \boxplus m$. The length m represents the case when no null character is found before the end of the buffer. This case would halt the program and is thus eliminated from the result. Then, primitive strcpy updates the destination buffer with the new abstract length. When the destination address a is precisely known, the information is replaced by the new abstract length bounded by the size of the source zone. When the possible destination addresses are contained in a buffer $(a : n)$, $\mathsf{weakstrcpy}$ merges the previous length with interval $[l; u + n - 1]$ bounded by the size of the destination zone. The lower bound l corresponds to the case when the smallest string is copied to a. The upper bound $u + n - 1$ corresponds to the case when the longest string is copied to $a \boxplus (n - 1)$. Notice how both primitives make extensive use of the algorithm Φ to change partitions. They satisfy conditions:

$$\left\{ l \;\middle|\; \begin{array}{l} \sigma \in \gamma(S) \wedge a \in \gamma_{\mathbb{A}}(A) \wedge n = \mathsf{allocsz}_{\mathcal{A}}(a) \\ l = \mathit{strlen}_\sigma(a : n) \wedge l \neq n \end{array} \right\} \subseteq \gamma_{\mathbb{Z}}(\mathsf{strlen}(S, A))$$

$$\left\{ \sigma[a \boxplus j \mapsto c_j]_{0 \le j \le l} \;\middle|\; \begin{array}{l} \sigma \in \gamma(S) \wedge a \in \gamma_{\mathbb{A}}(A) \wedge l \in \gamma_{\mathbb{Z}}(L) \\ n = \mathsf{allocsz}_{\mathcal{A}}(a) \wedge l < n \\ \forall 0 \le j < l : c_j \neq \text{'\textbackslash 0'} \wedge c_l = \text{'\textbackslash 0'} \end{array} \right\} \subseteq \gamma(\mathsf{strcpy}(S, A, L))$$

This ensures the soundness of the abstract string copy with respect to its concrete counterpart. Theorem 1 still holds.

4.3 Checks

Information gathered by the static analysis is used to check that all potentially dangerous memory manipulations are safe. A predicate is applied to the abstract value computed for the arguments of each operation. If the predicate does not hold, then the tool has insufficient information to conclude the operation is safe and it emits a warning. We present three such predicates[1]:

- *Buffer overflows:* when accessing an array of size n at any index in $\gamma_{\mathbb{Z}}(I)$, the index should be within bounds:

$$\mathsf{check}_{[]}(I, n) = (0 \le \min(I)) \wedge (\max(I) < n)$$

- *Pointer overflows:* when dereferencing a pointer P to a data of type τ, the pointer should be within the referenced zone:

$$\mathsf{check}_*(\langle A, I, N \rangle, \tau) = (0 \le \min(I)) \wedge (\max(I) + \mathsf{sizeof}(\tau) \le \min(N))$$

[1] all abstract arguments are suppose to be different from $\bot$.

– *String buffer overflows:* when copying a string of length l in $\gamma_{\mathbb{Z}}(L)$ from a source address a in $\gamma_{\mathbb{A}}(x, O)$ to some destination address a' in $\gamma_{\mathbb{A}}(y, O')$, there should be a null character before the end of the allocated memory starting in a and there should be at least l bytes of allocated memory from a':

$$\begin{aligned}
&\mathsf{check}_{\text{'}\backslash 0\text{'}}(S, (x, O), (y, O'), L) = \\
&\ \mathsf{tobuff}(x, O) \subseteq \mathcal{A} \wedge a = (x, \max(O)) \wedge m = \mathsf{allocsz}_{\mathcal{A}}(a) \wedge m \notin \Phi S(a : m) \\
&\ \ \wedge \mathsf{tobuff}(y, O') \subseteq \mathcal{A} \wedge \max(L) < \mathsf{allocsz}_{\mathcal{A}}(y, \max(O'))
\end{aligned}$$

5 Experiments

The static analysis was implemented in OCaml [16]. It uses CIL [18] as front-end. A simplification phase is applied to the CIL output to get to our kernel language. The analysis then propagates the abstract store following the structure of the code. Loops are dealt with simple fixpoint computation algorithms. Some loops are unfolded in order to improve precision. Once computations have stabilized, an ultimate pass checks potentially dangerous operations and emits warnings. Excluding CIL, the whole source code totals approximately 4000 lines of code.

The design of this static analysis was constantly lead by software most similar to what is found on actual aeronautical products. It is interesting to note that, in these case studies, approximately 60% of calls to `strcpy` have a constant string as source argument. Another 25% are called with a source buffer that is initialized with a constant string. Experiments were performed on small benchmarks from this software base. We sometimes had to manually remove union types which are not handled by this analysis. Among others, all 63 calls to `strcpy` in a 3000 lines of code program were successfully checked. Here is a small example that embodies some of the more difficult cases the tool had to process:

```
typedef struct {              int main() {
  char* f;                      s a[2][2];
} s;                            s* ptr = (s*) &a[1];
char buf[10];                   init(ptr);
                                ptr = (s*) &a[0];
void init(s* x) {               strcpy(a[1][1].f, "strcpy ok");
  x[1].f = buf;                 strcpy(a[1][1].f, "strcpy not ok");
}                             }
```

The tool flags the second call to `strcpy`. Since it knows variable `x` and `&a[1]` are aliased, it deduces that `a[1][1].f` has size 10 and doesn't emit any warning for the first call. This example demonstrates that the integration of several C features in one tool are necessary to obtain sufficient precision.

6 Related work

The detection of buffer overflows in C programs is an active field of research and various approaches have been proposed.

Fuzzing is a testing technique that consists in hooking a random generator to the inputs of a program. If the program crashes then defects may be uncovered.

Smart fuzzing tools take advantage of the network protocols [1, 3] or file formats [23] expected by the software in order to exercise the code in more depth. However, testing can usually not be exhaustive. Tools like StackGuard [8], ProPolice [2], CRED [20] and other [28] are C compiler's extension that implement runtime protection mechanisms. For instance, StackGuard uses a canary to detect attacks on the stack. Unfortunately, these techniques incur a non negligible overhead: either by slowing down execution or using up memory. Light static analyzes may remove unnecessary checks and improve performances [17]. In the end, all these techniques just turn buffer overflows into denial-of-service attacks.

Static analyses can detect defects before execution of the code. Several tools [27, 25, 4, 15, 11, 30, 26, 12] sacrifice soundness to scalability or efficiency. Unsound tools include fast and imprecise lexical analyzer such as ITS4 [25]. BOON [26] and [12] both translate the verification problem into an integer constraint problem but ignore potential aliasing. Soundness is clearly mandatory in our context. ASTREE [7], Airac [14], CGS [24] are all sound tools based on abstract interpretation that aim at detecting all runtime errors in C code. ASTREE focuses on control command software without pointers, Airac on array out of bounds and CGS on dynamic memory manipulation. These approaches do not have any special treatment for strings, which is a potential source of imprecision in our case. CSSV [9] and the analysis of [22] are most close to our work. Like us, both adopt the abstraction pioneered in [26] of strings by their possible lengths. Unlike us, they use the expensive numerical domain of polyhedra. They handle dynamic allocation. Instead of incorporating value and pointer analysis together, both perform the pointer analysis separately. CSSV then translates the C program into an integer program. It needs function level annotations to produce precise results during a whole program analysis. CSSV can handle union types, albeit in a very imprecise way: each memory location has a size and any assignments of a value of different size sets the location to *unknown*. Interestingly, the abstraction in [22] associates the length of strings to pointers, rather than to the buffer where the string is stored. It seems difficult to extend the formalism in order to deal with more language features. In particular, two pointers are aliased when they have the same base address and length. This condition is clearly too restrictive and prevents the handling of multi-dimensional arrays or cast operations.

7 Conclusion

We have designed and implemented a new static analysis to check the correctness of all memory manipulations in C programs. It integrates several analysis techniques to handle pointers, structures, multi-dimensional arrays, some kinds of casts and strings. The analysis of strings is made as simple as possible thanks to transport operators that let tune the granularity of the abstraction. First experimental results are extremely promising, and the abstraction seems adequate to prove actual case studies correct. Further work will explore the semantics and abstractions necessary to deal with C union types with much precision.

References

1. Dave Aitel. The advantage of block-based protocol analysis for security testing. Technical report, Immunity,Inc., 2002.
2. A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, 2000.
3. Philippe Biondi. Scapy. http://www.secdev.org/projects/scapy/.
4. B. Chess. Improving computer security using extended static checking. In *IEEE Symposium on Security and Privacy*, 2002.
5. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*. ACM Press, 1977.
6. P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1), 1979.
7. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *Proceedings of the European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*. Springer, 2005.
8. C. Cowan and al. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*. USENIX Association, 1998.
9. Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. ACM Press, 2003.
10. Mark W. Eichin and Jon A. Rochlis. With microscope and tweezers: An analysis of the internet virus of november 1988. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1989.
11. David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1), 2002.
12. Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM conference on Computer and communications security*. ACM Press, 2003.
13. Erich Haugh and Matthew Bishop. Testing C programs for buffer overflow vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2003.
14. Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In *Static Analysis, 12th International Symposium*, volume 3672 of *Lecture Notes in Computer Science*. Springer, 2005.
15. D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.
16. X. Leroy, D. Doliguez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system release 3.06, documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique (INRIA), 2002.
17. George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Transactions Programming Languages and Systems*, 27(3), 2005.

18. George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proceedings of Conference on Compiler Construction*, 2002.
19. Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.
20. Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Network and Distributed System Security Symposium*. The Internet Society, 2004.
21. Michael Siff, Satish Chandra, Thomas Ball, Krishna Kunchithapadam, and Thomas W. Reps. Coping with type casts in C. In *7th European Software Engineering Conference*, volume 1687 of *Lecture Notes in Computer Science*. Springer, 1999.
22. Axel Simon and Andy King. Analyzing string buffers in C. In *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*. Springer-Verlag, 2002.
23. Michael Sutton and Adam Greene. The art of file format fuzzing. In *Black Hat USA 2005*, 2005.
24. Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for large embedded C programs. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*. ACM Press, 2004.
25. John Viega, J. T. Bloch, Y. Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *16th Annual Computer Security Applications Conference*. IEEE Computer Society, 2000.
26. David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2000.
27. John Wilander and Mariam Kamkar. A comparison of publicly available tools for static intrusion prevention. In *7th Nordic Workshop on Secure IT Systems*, 2002.
28. John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Network and Distributed System Security Symposium*. The Internet Society, 2003.
29. Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, 1993.
30. Yichen Xie, Andy Chou, and Dawson R. Engler. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering*. ACM, 2003.
31. K. Yi. Yet another ensemble of abstract interpreter, higher-order data-flow equations, and model checking. Technical Memorandum 2001-10, Research on Program Analysis System, National Creative Research Center, Korea Advanced Institute of Science and Technology, 2001.