Security Audit using Extended Static Checking: Is It Cost-effective Yet?

***Abstract*—This paper describes our experience of doing variation analysis of known security vulnerabilities in C++ projects including core operating system and browser COM components, using an extended static checker HAVOC-LITE. We describe the extensions made to the tool to be applicable on such large components, along with our experience of using an extended static checker in the large. We argue that the use of such checkers as a configurable static analysis in the hands of security auditors can be quite cost-effective tool for finding variations of known vulnerabilities. The effort has lead to finding and fixing around 70 previously unknown security vulnerabilities in over 10 millions lines operating system and browser code.**

I. Introduction

Ensuring security of software has become of paramount importance to the software industry. Every software development group, representing either a small team of developers or an entire company, mandates extensive testing and analysis to safeguard against security breaches. However, security flaws will remain a part of life, at least in the case of legacy applications that cannot be redesigned from scratch. In such cases, effective defense mechanisms are required to mitigate the impact of security vulnerabilities. In particular, finding all possible variants of a known security vulnerability can go a long way in safeguarding the known attack surface of a software system.

The most common security practices in an industrial settings can be classified as either (a) *fuzzing* based, or (b) *static analysis*, or (c) *manual audit*. The fuzzing based or testing based methods rely on mutating existing tests (good or failure inducing) to test for failures, typically in the form of crashes. However, it is quite difficult to obtain high coverage (in such system-wide testing scenarios) of the attack surface, especially when the components are deeply nested. Static analysis practices consist of finding a few "patterns" that match the known vulnerabilities and looking exhaustively for such patterns in the source code with the aid of custom methods. Such tools make ad-hoc assumptions about the code to hide sources of false alarms, and therefore are not very effective in inspiring confidence of the bugs missed. Although static analysis for certain well-defined properties (such as certain classes of buffer overruns) have been effective at reducing one source of attacks [HDWY06], these tools are not agile enough to adapt to a new class of security vulnerabilities. Therefore, most of the work of ensuring security is relegated to manual code review around the known attack surface. However, manual reasoning without tools scales poorly with the need for inter-procedural inspection. Although there has been a lot of progress in deploying rigorous formal techniques to ensure the safety of realistic code-bases (e.g. [KAE+10]), it is far from being cost effective to be useful on real code-bases in the hands of average security testers.

Extended static checking tools (such as ESC/-Java [FLL+02a], HAVOC [BHL+10]) offer a potential to develop *configurable static analysis* tools with high coverage guarantees. These tools provide the user to write contracts (specifications of procedures) in the form of preconditions, postconditions, assertions and discharge them using modern Satisfiability Modulo Theories (SMT) solvers [Sat]. The semantics of the source language is precisely defined once by the tool (and does not vary by the property being checked), and the assumptions are well documented. Unlike full functional correctness verifiers (such as VCC [DMS+09]), they make pragmatic assumptions to reduce the complexity of proofs. Such tools are also equipped with simple yet robust user-guided contract inference tools (such as Houdini [FL01]) to reduce the manual overhead of writing specifications. Although the use of extended static checkers had been proposed for ensuring security a decade back [Che02], not much success has been reported in practical usage. Our conjecture is that the absence of a usable, robust and scalable tool for the space of core operating system and browser implementations has been one of the main causes for the lack of adoption.

In this paper, we present a case study of using an extended static checker HAVOC-LITE[1] for checking variants of security vulnerabilities in Microsoft Windows and Internet Explorer. We document the challenges encountered in deploying the previous version (henceforth called HAVOC) and the extensions needed to apply the tool in a realistic setting. This includes modeling most common C++ language features used typically in such applications, scaling the inference to be applicable to modules with several hundred thousand procedures, and early annotation validation. We then present our experience of security engineers using the tool on several properties devised as a response to a several existing vulnerabilities. Over the course of one year involving 3 security engineers, the effort led to discovering and fixing around 70 previously unknown security vulnerabilities in over 10 million lines of production code. We discuss the effort involved in modeling, performing inference of annotations and false alarms encountered along the way. We conclude that in spite of the current limitations,

[1]HAVOC-LITE is the new version of HAVOC [BHL+10] developed to meet the needs of this deployment.

such a tool can be (is already is) quite cost-effective in improving the productivity of security auditors who invest in manual audit or implementing ad-hoc tools to look for a few patterns.

The rest of the paper is organized as follows:

1) In Section II, we present an overview of our approach using two simplified examples. The examples illustrate some of the challenges posed when analyzing low-level systems code, a brief summary of the modeling effort, dealing with object-oriented features and the use of annotation inference.
2) In Section III, we provide a brief description of the existing tool HAVOC that has been applied to large Windows modules to find errors. In Section IV, we describe the main shortcomings of HAVOC that limited its application for our problem domain. We describe the design of HAVOC-LITE that includes modeling an expressive subset of C++ features, scaling the annotation inference by using a two-level algorithm that avoids memory blowup for large modules, along with other features required for making the tool robust and usable in the hands of security auditors.
3) In Section V, we describe the effort of applying HAVOC-LITE on the core OS and browser components. We discuss the properties that were modeled as variants of existing security vulnerabilities, candidate annotations required for inferring intermediate annotations, and some representative errors. We show the need for various pragmatic decisions (such as dealing with unsound modifies clause) to trade off soundness for cost-effectiveness of the analysis. We highlight the effort required in devising the inference to reduce false alarms and the payoff over the different properties.
4) In Section VI, we compare our effort with the work of other static analysis tools, and finally conclude in Section VII.

II. Motivating examples

In this section, we introduce two concrete examples containing commonly found programming style in C++ and COM (Component Object Model [Com]) applications. These examples can only be precisely analyzed if the semantics of bit-level manipulations are properly modeled, the common object-oriented and interface-oriented programming are well handled by the extended static checker.

A. Example 1: Interprocedural and bit-precise reasoning

The first example shows a generic container data structure called a `VARIANT` and commonly used in C++/COM applications. This structure contains a special field `vt` and a union of data attributes. The value of the `vt` field indicates which union field is correctly initialized. Failure to check the value of the `vt` field can lead to using the wrong union field and therefore use a pointer field containing an integer value. Such mistake is likely to lead to a security vulnerability (such as in functions `t1bad`). Most of the time, checking the `vt` field and using the associated union field are done in separate functions (such as in `t2good` and `t2good2`). Therefore, constraints must be adequately propagated inter-procedurally to avoid false positives from static analysis warnings. In addition, since the `vt` attribute is a bit field, bit-level program semantics needs to be supported by the static checker. Finally, a user needs to be able to document the desired property by checking each dereference of the set of fields (such as `pbarray`, `pbVal`)

```
1  typedef struct tagVARIANT {
2      struct __tagVARIANT {
3          VARTYPE vt;
4          union {
5              ...
6              SAFEARRAY *parray;
7              BYTE *pbVal;
8              ...
9              PVOID byref;
10             ...
11         };
12     };
13 } VARIANT;
14
15 bool t1good() {
16   VARIANT v;
17   v.vt = VT_ARRAY;
18   v.parray = 0;
19   return true;
20 }
21
22 bool t1bad() {
23   VARIANT v;
24   v.vt = VT_ARRAY;
25   v.pbVal = 0;
26   return true;
27 }
28
29 bool t2good() {
30   VARIANT v;
31   v.vt = VT_BYREF | VT_UI1;
32   func_use_vfield(&v);
33   return true;
34 }
35
36 void func_use_vfield(VARIANT *v) {
37   v->pbVal = 0;
38 }
39
40 bool t2good2() {
41   VARIANT v;
42   func_set_vt(&v);
43   v.pbVal = 0;
44   return true;
45 }
46
47 void func_set_vt(VARIANT *v) {
48   v->vt = VT_BYREF;
49   v->vt |= VT_UI1;
50 }
```

Fig. 1. Example of analysis requiring inter-procedural bit-level reasoning

under consideration. Such features are supported by HAVOC-LITE and was used to find multiple security vulnerabilities in a critical browser component.

```
1  #include <windows.h>
2  #include "havoc.h"
3
4  /* Field instrumentations */
5
6  __requires(v->vt == VT_ARRAY)
7  __instrument_write_pre(v->parray)
8  void __instrument_write_array(VARIANT *v);
9
10 __requires(v->vt == (VT_BYREF|VT_UI1))
11 __instrument_write_pre(v->pbVal)
12 void __instrument_write_pbval(VARIANT *v);
13
14 /* Func instrumentations with candidates */
15
16 __cand_requires(v->vt == (VT_BYREF|VT_UI1))
17 __cand_requires(v->vt == VT_ARRAY)
18 __cand_ensures(v->vt == (VT_BYREF|VT_UI1))
19 __cand_ensures(v->vt == VT_ARRAY)
20 __instrument_universal_type(v)
21 __instrument_universal_include("*")
22 void __instrument_candvariant(VARIANT *v);
```

Fig. 2. Annotations for the first example.

Figure 2 shows the annotations written by the user to create a checker for this property. There are two parts to the annotations: (a) devising the property and (b) creating an interprocedural inference.

The "Field instrumentations" are instrumentations provided to insert an assertion at reads to particular fields. For example, the instrumentation provided using `__instrument_write_array` method (the name of the method can be arbitrary except for the prefix `__instrument_`) specifies that every write to the field `parray` in the structure `VARIANT` is preceded by ("`write_pre`") an assertion that the `vt` field in the structure equals the value `VT_ARRAY`. There is a similar check before writes to the `pbVal` field using the instrumentation `__instrument_write_pbVal`. These two instrumentations allow the user to document the property to be checked.

We now look at how the user can configure the interprocedural annotation inference. The "Func instrumentations" are used to write annotations on a set of methods, instead of listing each individual method. The instrumentation primitive `__instrument_universal_type` specifies that any method that takes a parameter of type `VARIANT *` is instrumented. The filter `__instrument_universal_include` can be used to restrict the instrumentation to only methods whose names match a particular pattern — in this case, the wild-card pattern "*" matches any method. The annotations in `__cand_requires` and `__cand_ensures` are *candidate* preconditions and postconditions. These candidates are fed to the interprocedural annotation inference engine that infers a subset of them as annotations that actually hold on every context.

When the user runs HAVOC-LITE with the property and candidate annotations, the tool infers two annotations: The `func_use_vfield` has a precondition

```
__requires(v->vt == (VT_BYREF | VT_UI1))
```

and the method `func_set_vt` has a postcondition

```
__ensures(v->vt == (VT_BYREF | VT_UI1))
```

These additional annotations allow the tool to only complain about the method `t1bad`, which corresponds to the only true error. Although for this simple example, it is easier to simply write the two additional annotations, it is immensely useful when dealing with modules with several thousand deeply nested procedures.

B. Example 2: Object-oriented reasoning

A second example involves the `IUnknown` interface class which is responsible for all the reference counting mechanisms necessary to maintain object consistency in COM applications. In this example, the class method `A::Action` performs a call to `QueryInterface` which is in charge of retrieving an instance of the interface given an input interface identifier. Such a call performs an implicit call to `AddRef` which increases the reference counter for this interface. Success of this call (when the return status is `S_OK`) leads to calling method `WebAction` which performs the expected operation on class `B`. Failure to retrieve the interface methods leads to early termination, where the `IUnknown` interface pointer is released using a call to method `ReleaseInterface`, which is in charge of decrementing the reference count for this interface (provided the interface pointer is non-NULL).

A security vulnerability exists in this example due to the lack of NULL initialization of the `IWebIface` pointer, which leads to corrupting the reference counter of an unknown location in the program in case the call to `QueryInterface` is not successful. Such example requires an accurate object-oriented awareness from the static checker. We later show how HAVOC-LITE was used to uncover multiple similar security vulnerabilities in a critical browser component.

To model this property, the user can introduce a ghost field `Queried` in every object — the ghost field tracks whether an object has been created by a successful call to `QueryInterface`. The value of the ghost field for an object `x` is written as `__resource(``Queried'', x)`. One can write a precondition for the `Release` methods of `B` and any of its derived classes:

```
#define QUERIED(x)                          \
        __resource(``Queried'', x) == 1  \

__requires(QUERIED(this))
ULONG B::Release();
```

```
1  class A {
2    A() { };
3    HRESULT Action();
4    B       *Lookup();
5  };
6  class B : public IWebIface,
7            public IUnknown {
8    B() { };
9    HRESULT QueryInterface(IID id, void **p);
10   ULONG   AddRef();
11   ULONG   Release();
12   HRESULT WebAction();
13 };
14 void ReleaseIface(IUnknown *i) {
15   if (i) i->Release();
16 }
17
18 int main(int ac, char **av) {
19   A *a = new A();
20   B *b = a->Lookup();
21   a->Action(b);
22   return (0);
23 }
24
25 HRESULT A::Action(B *b) {
26   HRESULT r = S_FAIL;
27   IWebIface *w;
28   if (b == NULL)
29     goto Cleanup;
30   r = b->QueryInterface(IID_WebIface, &w);
31   if (r == S_OK)
32     r = b->WebAction();
33 Cleanup:
34   ReleaseInterface(w);
35   return (r);
36 }
```

Fig. 3. Example of analysis requiring precise object-oriented semantics

to indicate that the receiver object of the method `Release` has to be created by an earlier call to `QueryInterface`. This is in turn modeled by writing the following postcondition for the method:

```
__ensures(__return != S_OK || QUERIED(*p))
HRESULT B::QueryInterace(IID id, void **p);
```

where `__return` denotes the return value of the procedure.

Finally, one needs to infer annotations such as:

```
__requires(i == NULL || QUERIED(i))
void ReleaseIface(IUnknown *i);
```

which can be done with the help of the first populating candidate annotations on all methods that consume a `IUnknown` as an argument (We do not show the instrumentation here), and then performing interprocedural annotation inference.

III. Background: HAVOC

In this section, we provide a background on HAVOC. We describe HAVOC-LITE along with the extensions created for this paper in Section IV.

HAVOC can be best thought of as an extended static checker for C programs, in the spirit of ESC/Java [FLL+02b]. It provides an (a) extensible property checker with the aid of an annotation/contract language, (b) a procedure modular verifier that provides an accurate depiction of C semantics, (c) an user-guided inter-procedural annotation inference engine, along with (d) various instrumentation primitives. Figure 4 shows the overall usage model of HAVOC (and also of HAVOC-LITE) in the hands of a user. We describe each of the components briefly in the next few subsections. More details about these features can be found in an earlier work [BHL+10].

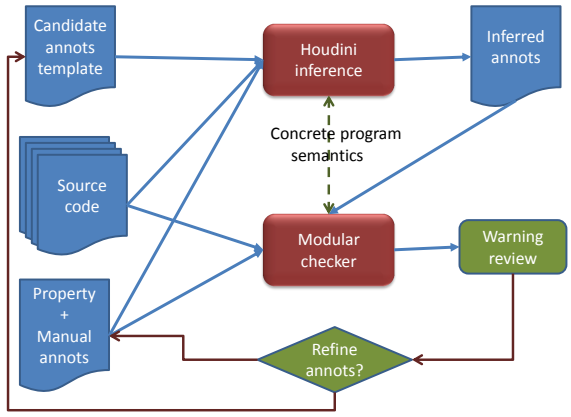

Fig. 4. HAVOC and HAVOC-LITE flow.

A. Contract language

A user can document contracts about the code using an annotation language. An annotation is an assertion over the state of the program. There are four classes of assertions (i) assertion `__assert` (e), (ii) assumptions `__assume` (e), (iii) preconditions `__requires` (e) and (iv) postconditions `__ensures` (e). Here e refers to a side-effect-free C expression that evaluates to a scalar or pointer value — in other words, e cannot be a structure value. For example, one can write `__ensures (*x == y->f)` to indicate that on exit from a procedure the value obtained by dereferencing the variable `x` is identical to the value stored in a field `f` inside the structure pointed to by `y`. In addition to assertions, a user can specify a `__modifies` clause that specifies which part of the heap is modified by a procedure. For the purpose of this paper, we ignore discussion on loop invariants, which are also supported in HAVOC.

In addition to the expressions in scope, a user can refer to the return variable by using the symbol `__return` in a postcondition, refer to the value of an expression at entry to a procedure using `__old ()` in a postcondition. For example, `__ensures (__return == __old (*x) + 1)` signifies that the value of return variable is one more than the value stored in `*x` at the entry to the procedure. The user can also refer to the state of *ghost fields* that augment the state

$$
\begin{array}{lrcl}
\text{Locs} & l & ::= & *e \mid e \rightarrow f \\
\text{Expr} & e & ::= & x \mid n \mid l \mid \&l \mid e_1 \texttt{ op } e_2 \mid e_1 \oplus_n e_2 \\
\\
\text{Command} & c & ::= & \texttt{skip} \mid c_1; c_2 \mid x := e \mid l := e \mid \texttt{if } e \texttt{ then } c \mid \texttt{while } e \texttt{ do } c \\
 & & \mid & f(e, \ldots, e)
\end{array}
$$

$$
\begin{array}{lcl}
E(x) & = & x \\
E(n) & = & n \\
E(e \rightarrow f) & = & \mathsf{Mem}^f[E(e) + \mathsf{Offset}(f)] \\
E(*(e:\tau)) & = & \mathsf{Mem}^\tau[E(e)] \\
E(\&e \rightarrow f) & = & E(e) + \mathsf{Offset}(f) \\
E(\& * e) & = & E(e) \\
E(e_1 \texttt{ op } e_2) & = & E(e_1) \texttt{ op } E(e_2) \\
E(e_1 \oplus_n e_2) & = & E(e_1) + n * E(e_2)
\end{array}
$$

$$
\begin{array}{lcl}
C(\texttt{skip}) & = & \texttt{skip} \\
C(c_1; c_2) & = & C(c_1); C(c_2) \\
C(x := e) & = & x := E(e); \\
C(l := e) & = & E(l) := E(e); \\
C(\texttt{if } e \texttt{ then } c) & = & \texttt{if } (E(e)) \ C(c) \\
C(\texttt{while } e \texttt{ do } c) & = & \texttt{while } (E(e)) \texttt{ do } C(c) \\
C(f(e_1, \ldots, e_k)) & = & \texttt{call } f(E(e_1), \ldots, E(e_k))
\end{array}
$$

Fig. 5. A simplified subset of C, and translation from C into BoogiePL. $E()$ maps a C expression into BoogiePL expression and $C()$ maps a C statement into a BoogiePL statement.

of the program using a keyword `__resource`[2]. The scalar expression `__resource` (s, e) where s is a string and e is a side-effect pointer/scalar expression, refers to the value of a ghost field named "s" inside the structure pointed to by e; i.e. the value of `e->s`. The user can modify such ghost fields in the program.

The annotation language, along with the presence of ghost fields allows user to encode various interesting properties of the program. Section II provides a few examples of such properties. Several more examples of properties can be found in earlier works [BHL$^+$10].

HAVOC provides a sufficiently accurate memory model for C programs that provides meaning to constructs such as pointer arithmetic, casts, yet supports common disambiguation required for scalable reasoning of high level properties. Figure 5 provides a simplified subset of C (for illustration purposes) without nested structures and addresses of variables. Figure 5 also provides the translation of this subset into an intermediate verification language BoogiePL [DL05]. BoogiePL is a simple procedural language, where the set of variables are restricted to Boolean, integers and arrays over them. The operator $C()$ maps a C statement to the equivalent BoogiePL statement in a straightforward manner. In addition, the language has support for assertions, assumptions, preconditions and postconditions — the HAVOC annotations map directly to them.

The operator $E()$ maps a C expression (present in either a statement or an annotation) into a BoogiePL expression. The heap is split into a finite number of arrays (named $\mathsf{Mem}^f[]$ or $\mathsf{Mem}^\tau[]$), one per scalar field or pointer field (f) or pointer type (τ). Dereferencing a pointer is modeled as indexing into the appropriate array with a suitable offset — the operator $\mathsf{Offset}(f)$ provides the offset of the field f inside its parent structure. The heap splitting assumes *field safety* [CHLQ09] that allows exploiting the types and fields in the program to get disambiguation. Under field safety, it is assumed that $\&x \rightarrow f$ can never alias with $\&y \rightarrow g$ for distinct field names f and g. Further, addresses of a field $\&x \rightarrow f$ does not alias with the address $\& * e$. Although HAVOC has an option to not assume field safety, the annotation overhead increases several fold even for simple examples.

[2]Ghost variables are also supported as a degenerate case of ghost fields.

B. Modular verifier

An annotated BoogiePL program is checked for correctness one procedure at a time by the Boogie program verifier. The verifier uses *verification condition generation* (translation of the program into a logical formula with near linear size) [BL05] and automated theorem proving (namely Satisfiability Modulo Theories solvers [Sat]) to check the satisfiability of the formula. HAVOC lifts an intraprocedural counterexample at the BoogiePL level to display over the C source code.

C. User-guided inference

HAVOC uses a variant of the Houdini algorithm to perform inter-procedural inference of procedure annotations [FL01]. A user can write a set of *candidate* preconditions (`__cand_requires` (e)) and postconditions (`__cand_ensures` (e)) in addition to the usual annotations. The Houdini algorithm performs an inter-procedural greatest fix-point analysis to retain the (unique) maximum subset of these candidates that can be proved modularly by the program verifier, while assuming the non-candidate annotations. The fix-point proceeds by maintaining a worklist of procedures to be checked. At each step, a procedure p is checked using the modular verifier. Any candidate annotation that cannot be proved is removed from the list of annotations. Depending on the nature of the removed candidate, either the callers of p (for candidate postcondtiions), or the callee q (whose candidate preconditions are removed) are added to the worklist, in addition to p. The process is repeated until the worklist is empty. The simple algorithm terminates in at most $n * c$ iterations, where n is the total number of procedures in the module, and c is the total number of candidate annotations.In

practice, it runs almost linear in c, thus guaranteeing a quick turnaround.

D. Instrumentations

Finally, various syntax-based instrumentation facilities are provided to avoid manually writing annotations on large code-bases. For example, an user can instrument all procedures whose names match a regular expression with a postcondition on globals, or instrument all procedures that take a parameter x of type $\tau*$ by a precondition parameterized by x. Moreover, these annotations can include candidate annotations as well — this is crucial to performing the annotation inference. In addition to procedure annotations, the user can also instrument reads and writes to specific fields, types or globals to insert an assertion or assumption before or after the source line (as illustrated in Section II).

IV. HAVOC-LITE

The following were the main limitations of HAVOC in terms of usability and applicability for the code bases under investigation. In the process of making the tool more robust, we dropped support for some of the earlier features in HAVOC (hence the name HAVOC-LITE) such as checking type-safety of C programs, dealing with linked list invariants and support for complex modifies clauses in candidate annotations described in earlier works [LQR09].

A. Modeling C++ language constructs

By far, the most significant shortcoming of HAVOC was inability to deal with most C++ constructs. In this section, we briefly describe some of the changes required to handle the most common C++ features used commonly in the COM components. We illustrate the translation of C++ to BoogiePL program informally with the aid of a simple example in Figure 6 and the translated BoogiePL program in Figure 7.

To handle *instance methods*, we make the receiver object explicit by exposing the `this` pointer. The BoogiePL translation of the procedure `C::f` takes `this` as an argument, and updates the field array `Mem_e` that represents the field `e` in the class. We use the function `Offset_f_C` to denote the offset of a field `f` in a class `C`. Further, the annotation expressions can refer to `this` in assertions, as shown for this method.

Constructor calls are modeled by first allocating a contiguous buffer with the size of the class by invoking the special method `__HV_malloc`. The mutable variable `alloc` monotonically increases to ensure that the buffer allocated is fresh.[3] The constant map `Base` tracks the base of the buffer for any pointer in the buffer, and the constant map `DT` maps the base of the buffer to its dynamic type (the type used during allocation). The constraints are enforced by the specification of the procedure `__HV_malloc` specified using the `__ensures` annotations. The newly allocated object is then passed to the constructor of the class.

Dynamic casts is modeled by a conditional assignment that checks if the right hand side (RHS) of the assignment is null or not. In case, the RHS is non-null, it assigns the pointer shifted by the offset of the base class in the derived type. The functions `Delta_A_B` model the offset of a base class `A` inside a derived class `B`.

Field access is modeled similar to C programs as before, except for accessing fields inside base classes. To access fields in a (transitive) base class (such as the field `a` defined within the base class `D` from within `B`), we first add the offset of the base classes in the access path from the derived class (`Delta_D_B` for this example).

Non-virtual method calls are handled similar to C, except the addition of the `this` parameter. Similar to field access, calling a method in one of (transitive) base classes require the addition of the offsets of the base classes in the access path. *Virtual method calls* require looking up the dynamic type (stored in `DT`) of the start of the object into a temporary variable `base` and performing a case split on the possible set of runtime types. For each dynamic type, the corresponding

```
1  class D {
2   public:
3        int a,b;
4        virtual void f(){}
5  };
6  class A : public D {
7   public:
8        int c;
9        A();
10       void f();
11 };
12 class B : public D {
13  public:
14      int d;
15      B();
16      void f();
17      void h();
18 };
19 class C : public A, public B {
20  public:
21       int e;
22       C();
23       void f();
24 };
25 ...
26 /* postcondition */
27 __ensures (this->e > old(this->e))
28 void C::f(){e++;}
29 ...
30 int main() {
31   C* pc = new C(); /* allocation */
32   B *pb = (B*) pc; /* dynamic cast */
33   D *pd = (D*) pb; /* dynamic cast */
34
35   pc->e = 4;        /* field access*/
36   pb->a = 6;        /* base field access*/
37
38   pc->h();      /* method call   */
39   pd->f();      /* virtual method call */
40 }
```

Fig. 6. A simple C++ example.

[3] We currently do not model deallocation.

```
1  function Delta_D_A(x) {x+0}
2  ...
3  function Delta_B_C(x) {x+12}
4  ...
5  function Offset_e_C(x) {x+24}
6  ...
7  var alloc: int;
8  const Base: [int]int;                    /* pointer to the start of an object */
9  const DT:   [int]int;                    /* dynamic type */
10
11 modifies alloc
12 ensures (new == old(alloc))
13 ensures (alloc > old(alloc) + n)
14 ensures (forall u \in [new,new+n). Base(u) = new)
15 ensures (DT(new) == type)                /*set the dyn type*/
16 procedure _HV_malloc(n, type) returns new;
17
18 ensures (Mem_e[Offset_e_C(this) > old(Mem_e[Offset_e_C(this)]))
19 procedure C::f(this){Mem_e[Offsetset_e_C(this)]++;} /* explicit ''this'' */
20
21 procedure main() {
22   pc = _HV_malloc(28, C);                 /* 7 integer fields of 4 bytes*/
23   call C::ctor(pc);                       /* constructor call*/
24
25   pb = (pc == 0 ? 0 : Delta_B_C(pc));     /* cast */
26   pd = (pb == 0 ? 0 : Delta_D_B(pb));     /* cast */
27
28   Mem_d[Offset_e_C(pc)] = 4;              /* field access */
29   Mem_d[Offset_a_D(Delta_D_B((pb)))] = 6; /* field access of base class */
30
31   call B::h(Delta_B_C(pc));               /* method call */
32   /* virtual method call */
33   base = Base(pd);                        /* obtain the start of object */
34   switch(DT(base)) {
35      case A: call A::f(base);break;
36      case B: call B::f(base);break;
37      case C: call C::f(base);break;
38      default: call D::f(base);break;
39   }
40 }
```

Fig. 7. The BoogiePL (cleaned up) for the C++ example.

virtual method is invoked (assuming for simplicity the method is defined in each derived class). In each of the cases, the pointer passed as the `this` parameter is the start of the object stored in the map `Base` during allocation. Currently, we assume a separate analysis to compute the set of potential target types that is fed into HAVOC-LITE as an input.

In addition, HAVOC-LITE also handles *operator overloading*, simple forms of *templates* (both parameterized by types or values) and other C++ features commonly encountered in COM. Although the modeling is far from being considered complete for C++, it allows us to get substantial coverage of many C++ code-bases using COM.

B. Instrumentations

The instrumentation mechanism in HAVOC was extended to support some object-oriented constructs. These include (a) instrumenting all instance methods declared in a given class, and (b) instrumenting an instance method in all classes.

The first feature is useful for annotating *class invariants*, by instrumenting all instance methods in a class with a precondition and postcondition. In addition, the user can remove the constructors and destructors from the set by using a set of patterns that are excluded. For example,

```
__requires(x->f != null)
__ensures (y->f != null)
__instrument_universal_exclude(``A$dtor'')
__instrument_universal_exclude(``A$ctor'')
__instrument_universal_include(``A$*'')
__instrument_universal_type(x)
void __instrument_class_inv(A *x);
```

instruments all methods in the class `A` (denoted by the `_include` pattern) except the constructor and destructor (denoted by the `_exclude` patterns) require and ensure that a field `f` is non-null for the receiver ("this") object. Instance methods that take additional objects of type `A` or static methods that take an object of type `A` have to be manually excluded though.

The second feature is often useful for getting the effect of *annotation inheritance*, where annotation on a virtual method is inherited by the overriding methods of all derived classes.

All instance methods with the name `foo` can be specified with the pattern `''*$foo''`. However, we currently do not provide any support for checking that derived classes only weaken preconditions and strengthen postconditions — it is left to the user to enforce.

C. Inference

HAVOC used the Houdini algorithm [FL01] to choose inductive invariants from a set of candidates. Although the inference was successful in scaling to modules with several hundred to a few thousand procedures [BHL$^+$10], it did not scale to the modules that had several hundred thousand procedures measuring several million lines of code. In this section, we discuss the improvements made to make the inference scalable to these modules.

1) Persisting fewer Boogie files: The approach in HAVOC generated a single BoogiePL file on disk containing the definition of all the procedures, and then invoked the Houdini procedure inside Boogie. However, creating a Boogie file with almost a million procedure did not scale as HAVOC crashed due to memory blowup during the generation of the Boogie file, and Boogie could not load such a file and perform VC generation. Instead, we first changed the flow to generate a Boogie file per procedure, and fed Boogie a list of Boogie files. This avoided the memory blowup in HAVOC, but still caused Boogie to take a long time.

Our first observation that for the purpose of Houdini inference, one can safely filter procedures that do not have any (a) candidate assertions inside the body of the procedure, (b) does not have any candidate postconditions, (c) does not have (immediate) callees that have a candidate precondition. This is because analyzing such procedure will always return "verified" as there are no assertions to check. This simple optimization allowed us to reduce the number of Boogie files used during the inference by at least 2 orders of magnitude, when the annotations were sparse.

2) Two-level Houdini algorithm: In spite in the dramatic reduction in the number of procedures being analyzed during Houdini, we were still left with several tens of thousand of procedures to analyze, and maintaining all the procedures (and their VCs) in memory during Houdini exceeded memory. To alleviate it, we designed a two-level Houdini algorithm that uses a cache to only pass a small set of procedures to the Houdini procedure.

Given a set of procedures P with candidate annotations C and a cache size n, the algorithm operates as follows.

1) It first initializes a work list W with all the procedures in P.
2) At each stage, it removes a set S of n procedures from W (or all procedures if $|W| \leq n$) and invokes Houdini on S.
3) Houdini removes a (possibly empty) subset of annotations $R \subseteq C$ after analyzing S.
4) For each removed candidate $c \in R$, we update the worklist as follows: (a) if c is a precondition of a method $p \notin S$, then we add p to W, and (b) if c is a postcondition of a method p then we add any of its caller q that is not present in S.
5) At the same time, C is updated to $C \setminus R$ by removing the candidates removed by Houdini.
6) The method is repeated until W is empty.

It is not difficult to show that when W is empty, then the set of candidates in C can be proved modularly.

The size of the cache influences the overall runtime as larger the cache, the more chance the (inner) Houdini algorithm gets to perform optimizations, and thus the outer loop converges faster. On the other hand, making the cache large increases the memory consumption for Houdini. We have observed that the memory requirement is really a function of the size of the procedures instead of the number of procedures. For a given problem where the size of the largest procedure is k, it is useful to set n so that $n * k$ does not exceed the memory alloted to the process. For most of our experiments, n is set between 20 and 100.

3) Candidates on roots and leaves: It is well known that if the set of candidate preconditions on *root* procedures (that have no callees) or the set of candidate postconditions on *leaf* procedures (that have no body) are inconsistent, then Houdini can infer annotations that may not hold. Previously, the users of HAVOC manually ensured that there root procedures have no candidate preconditions and leaf procedures have no candidate postconditions. This was an expensive process as determining the roots can be tricky when certain procedures are targets of function pointers. To simplify matters, HAVOC-LITE first removes candidate preconditions from the root and candidate postconditions from the leaves of the call graph presented to it for annotation inference. This substantially improves the usability of the inference for the user.

D. Other

In addition to the above enhancements, a number of other usability issues were addressed. First, we added a mechanism to insert annotations completely *non-intrusively* — i.e. the source tree did not have to be modified. Annotations were first compiled with type and procedure declarations and then linked with the definitions later. This greatly improved the adoption in teams that did not want to modify the source code, even to include an annotation header file. The separate compilation of the annotation file also allowed us to correct annotation parse errors quicker. HAVOC-LITE also added bit-vector support by interpreting the scalars and pointers as fixed-size bit-vectors, and using the theory of bit-vectors inside SMT solver Z3. Earlier implementation in HAVOC used unbounded integers and could not model the bitwise operations accurately. Finally, the user can associate custom strings and identifiers for warning messages that are displayed on assertion failures — this allowed easier triaging of warnings when multiple properties were checked in a single run.

V. Evaluation

In this section, we go through the evaluation of the capabilities of HAVOC-LITE to detect variations of known software

Properties	Description
Zero-sized allocations	Dynamic memory allocations should never be of size 0.
Empty array construction	There is always at least one element in new[] allocated arrays
VARIANT initialization	VARIANT structures should never be used without initialization.
VARIANT type safety	VARIANT union fields should never be referenced without proper field type value.
Interface reference counting	Interfaces should never be released without prior reference or initialization.
Library path validation	Dependences modules should never be loaded without fully qualified path.
DOM information disclosure	DOM accessors only returns success on successful completion

Fig. 8. Checked security properties

problems that are commonly reported in Microsoft products. We give to this methodology the name of *variation analysis*, capturing the intuition that a contract can be easily synthesized by a security auditor once a single security vulnerability of a kind is discovered. This contract is used to check *variations* of such vulnerability class across the whole code base.

A. Checked properties

We have evaluated the tool on the large scale on multiple heterogeneous properties across multiple products written in C and C++, summarized in Figure 8.

Some of the properties we describe in this table are in fact a family of properties. For example, zero-sized allocations is an umbrella denomination for calls to dynamic memory allocation APIs such as the user-mode *malloc*, the kernel-mode *ExAllocatePool*, dynamic array allocations via the *new* operator, or object constructors taking an integer parameter that is used to allocate an internal buffer for the class within this object class (e.g. the *CComBSTR* class allocates an array of bytes whose number of entries is the integer parameter of its constructor).

Other interesting properties come from expected contracts of special *VARIANT* object (recall the example from Section II). The type names have been changed to avoid unintended consequences of releasing such information. Box data structures can be the source of multiple security vulnerabilities if they are incorrectly initialized or used incorrectly.

Another interesting property ("Interface reference counting") arises from the need to enforce the usage of interfaces in object-oriented programs. An interface can be seen as a structure that holds a fixed list of function pointers. Interfaces are usually reference counted and released once their expected life time has been reached. Our introductory example in the overview section was taken from this class of security vulnerabilities affecting object-oriented software.

The property of library path qualification captures the intent that no binary dependence should be loaded from an unknown location under the threat of loading untrusted code (potentially from a remote location if the path is in the UNC format such as $\backslash\backslash remote\backslash machine\backslash untrusted.dll$). On the other hand, a preceding call to a trusted path-retrieving API such as *GetSystemDirectory* acts as a sanitizer, as it provides a proof that such path is prefixed by a string of the form $c:\backslash windows\backslash system32$.

B. Results

The result of checking those properties over the course of a year is presented in Figure 9. We identified more than a hundred vulnerabilities in critical software components using a build server equipped with 48 cores and 96GB of RAM to perform this analysis. While the big number of cores significantly speeds up the intra-procedural analysis, the inter-procedural does not currently benefit from it due to the sequential implementation. The "Check" time is the time to only check the annotations intraprocedurally, including inferred annotations if any. The "Inference" time is the time taken to perform the interprocedural annotation inference. Both times are reported in minutes. The table only reports 67 warnings for the properties mentioned. We also report a few other bugs for other properties later in this section — we did not perform a thorough evaluation of times etc. for these properties.

We only applied a particular property to the code bases that were affected by the property — this explains the difference in code size for each experiment. While some of the properties (e.g. Library path qualification) affected all user-mode code bases, others (such as the DOM property) only affects the core browser engine. Other generic COM properties such as the VARIANT initialization and type safety checking and the interface reference counting were ran on a set of large user-mode code bases making heavy use of such features. Two properties affecting the VARIANT structures were checked together on the same code-base, hence the check time and inference time are the same for both.

Note that inference information is not available for two properties: the library path qualification acted on tens of millions of lines of code, a size for which inference is not able to scale due to the sequential implementation. Likewise, the DOM information disclosure property only affected class accessors of the form *CBrowserObj*::*get_attr* totalizing only a few hundred methods to check in just 8 different classes for which the manual warning review process was fast enough without inference.

In addition to the properties described in Figure 8, we also uncovered security critical bugs from other checks. We describe a couple of them next.

We discovered a set of errors during the translation from C source files to the well-typed BoogiePL language. One of the checked kernel driver was using the following mechanism at

Properties	LOC	Procnum	Bugs	Check time	Inference time
Zero-sized allocations	2.8M	58K	9	3h14	3h22
Empty array constructor	1.2M	3.1K	0	26m	6m13
VARIANT initialization	6.5M	196K	5	5h03	11h40
VARIANT type safety	6.5M	196K	8	5h03	11h40
Interface reference counting	2M	11.2K	4	2h26	20h
Library path qualification	20M	Millions	35	5d	N/A
DOM information disclosure	2.5M	Hundreds	2	1h42	N/A

Fig. 9. Summary of results.

multiple locations:

```
void fctelm() { (...) }
int syscall(int num) {
    return ((*fctptrs[num])());
}
```

where *syscall* is a kernel entry-point returning an integer error code to the unprivileged user, and *fctelm* is a void-returning function part of a function pointer array *fctptrs* whose elements were supposed to return an integer. Such unsafe function pointer elements in the array were possible at compilation time due to the use of unsafe function pointer casts. Such defect can result in information disclosure security vulnerabilities since the return value (on the Pentium architecture, held in the *eax* register) is uninitialized in the *fctelm* function and used for another purpose (for example, holding the pointer to sensitive kernel data structures). We discovered around 30+ bugs in the driver that were fixed.

We also applied HAVOC (earlier version) to check User-kernel pointer probing on the Windows application APIs — to ensure that user-mode pointer should always be validated by specific `Probe*` APIs before being dereferenced in the kernel. The effort on around 300KLOC revealed another 7 bugs that were fixed.

C. Inter-procedural annotation inference

Inference is a useful technique to improve the checker's result when properties depend on the caller function context and the results of callee functions.

Figure 10 shows the difference of signal/noise ratio when the user-guided annotation inference was used. Inter-procedural analysis brings improvements in precision and signal/noise ratio, but the impact varies by the property. We found that for most sparse properties (such as the API related properties where only small numbers of checks are performed compared to the program size), the number of warnings diminishes by 10% to 45% depending on the checked property. A simple example of inference used to check the zero allocation property used generated candidate contracts of the form:

```
__cand_requires(param != 0)
__cand_ensures(__return != 0)
int fct(int param) { ... }
```

for every function accepting integer parameters and integer return values. Therefore, enforced preconditions of the form:

```
__requires(size != 0)
void* ExAllocatePool(unsigned int size);
```

can be checked with knowledge of the inferred (persisted) constraints at function boundaries. In this case, the burden of writing the candidate annotations is very small and brings great benefit to the signal/noise ratio before the checking phase is performed. We have applied this methodology to multiple properties as indicated in the table below:

Properties such as *interface reference counting* are ones for which inference is the most useful as the number of pointers to be tracked within the target modules is generally quite large. Such pointers can also sometimes passed between modules, which limit the ability of inference to filter out false positives. This is reflected in the signal-noise ratio of Figure 10. In general, our experiments support the fact that while inference does not suffice to reach a perfect analysis result, its use allow diminishing the burden of warning reviews in an appreciable way for security auditors.

Annotation inference works best when the set of intermediate annotations can be concisely expressed using candidate annotations that can be added with simple instrumentations. A user starts with the property under check, inspects the warnings from the checking and then devises a small set of candidates. Annotation inference is performed and the new set of warnings are noted and new candidates are added. The process remains cost-effective up to a couple of iterations, beyond which the auditor preferred to manually inspect the false alarms.

Cases where inference fails to improve the signal/noise ratio include functions with aliasing pointer parameters for which no type-state can be persisted without introducing more complex conditional constraints. In other cases, the annotations refer to fields of parameters, often not limited to a few. While such conditions can be encoded with modest manual effort for the inference engine to consume on individual functions, it is difficult to come up with generic instrumentations for writing these candidates on a module.

D. Use of unsound modifies clauses

In addition to using inference, we also performed an evaluation with the use of *modifies* clauses, where extra assumptions were added telling that the state of the heap did not change

Properties	Warns Warns	Warns with inf	Improv.	Cand.	Inf. cand.
Zero-sized allocations	71	50	29%	75162	42160
Empty array constructor	45	35	22%	4024	446
VARIANT initialization	216	117	45%	100924	770
VARIANT type safety	83	68	18%	100924	770
Interface reference counting	746	672 (3)	10%	234K	1671
DOM information disclosure	82	N/A	N/A	N/A	N/A
Library path qualification	280	N/A	N/A	N/A	N/A

Fig. 10. Results of running annotation inference. "Cand." denotes the number of candidate annotations and "Inf. cand." is the number of annotations that were inferred to hold.

when function calls are performed. This is an unsound assumption but has the advantage of bringing down the number of warnings to a very high signal/noise ratio. For example, when checking the Interface reference counting property, enabling this option brought the warning number down from 672 (after using inter-procedural inference) to 3, of which all were valid vulnerabilities. No extra annotations were explicitly necessary for such assumptions, as HAVOC-LITE provides an option through the configuration file. A similar decrease in warning numbers is witnessed on other properties though this option is only deemed necessary when the initial set of (false positive) alerts is big enough to justify losing soundness.

E. Cost-effectiveness and warnings review

The cost-effectiveness of using an extended static checker varies depending on each property. We found that security properties related to API calls are generally sparse in the sense that only a few calls are performed compared to the total number of lines of code and procedures of the analyzed modules. Checking field dereference also came with a reasonable return on investment as long as the number of fields that were checked for dereferences remained small enough. We found that the signal/noise ratio for sparse properties (such as library path qualification, zero-sized allocations, VARIANT initialization or VARIANT type-safety) was acceptable as we consistently found new vulnerabilities in the reported warning list. Other denser properties (such as the previous attempt of user kernel pointer probing) were harder to check with a signal-noise ratio of 5-10% . We explain this by the need to have every single pointer dereference be instrumented, representing tens of thousands of dereferences on medium sized modules.

Another interesting metrics to measure the success of such tool comes from the engineer feedback. We found that, on average, a security expert can review between 25 and 50 warnings per day. For some properties involving very deep inference and for which a function containing an alert has a large number of callees, we found that it can take one to four hours to review a single warning. This can also happen in case of a very complex control flow that arises within or between multiple components, often involving indirect call sites via function pointers or virtual methods, including call-back that sometimes cross a domain boundary (i.e. user-mode call-backs). For such instances of warnings, the help of a debugger is often necessary to understand whether the discovered vulnerable context is feasible. Since we did not use taint-analysis (a popular analysis to uncover user-controlled data dependences) such time also includes the discovery whether the variable values in the environment can be controlled by an attacker. Indeed, a software check may not be necessary if the variable values that can trigger the unsafe behavior are not under the control of the user (for example, the variables can be under control of a trusted third party components on which the analysis was not performed).

Overall, we found that the use of *HAVOC LITE* was cost-effective compared to pure manual code review. For example, we were able to guarantee the absence of unsafe integer overflows in a one million lines component after only 3 hours of warning review. A similar analysis done purely in a manual fashion would have required weeks if not months of work. The ability of static analysis to focus on crucial control locations and specific data manipulations was fundamental in this exercise. The ability to perform inter-procedural inference brought down the warning number consistently, saving hours of warning review to the analyst.

F. Found vulnerabilities

We now give a few examples of vulnerabilities that we were able to identify using extended static checking, starting with the interface reference counting vulnerability class.

1) Interface reference counting: The first example (Figure 11) of found and fixed vulnerability relates to the reference counting property of interfaces. The COM model makes heavy use of interface pointers, in particular in object-oriented projects like the browser where deep levels of inheritance are used and objects in different hierarchies share some functionalities. Those functionalities are therefore implemented in interfaces that classes can inherit without having to derive from another class. The *IUnknown* interface serves as a base interface for all other interface and class types (such as *CBrowserElement* in this example) by implementing three core methods called *QueryInterface*, *AddRef* and *Release*. A call to *QueryInterface* accepts an interface identifier and returns an array of function pointers that represents the implementation of the desired interface. If no such interface is available in the base object, *QueryInterface* will return an error. Otherwise, it

will return status code *S_OK*. Note that *QueryInterface* also performs a call to *AddRef* on the base class if the query is successful, so that the class is not freed while the program still holds a reference onto one of its interface. A critical safety property of such model states that any pointer on a COM interface written to via a call to *QueryInterface* should be released by a call to *ReleaseInterface* after usage, except when the call to *QueryInterface* failed (in that case, there is nothing to release). The *ReleaseInterface* API is simply a wrapper to the *Release* method that adds a wrapping check that ensures that the interface pointer is not NULL (in that case, the function is simply a NO OP).

```
1  CSomeElement::add(CBrowserElement *pElem){
2   HRESULT              hr;
3   IUnknown *           pUnk;
4   if (!pElem) {
5     hr = E_INVALIDARG;
6     goto End;
7   }
8   hr = pElem->QueryInterface(IID_ExpectedType,
9                              (void**) &pUnk);
10  ReleaseInterface(pUnk);
11  if (S_OK != hr) {
12    hr = E_INVALIDARG;
13    goto End;
14  }
15  hr = AddOption(pElem, pElem->str, FALSE);
16 End:
17  return (SetErrorInfo(hr));
18 }
```

Fig. 11. A real interface reference counting vulnerability (obfuscated)

In this example, this mechanism is used to guarantee that a given browser element *pElem* is of the intended type. However, the interface is never used as the looked up content is automatically discarded via the call to *ReleaseInterface* and the element is simply passed to method *AddOption* for storage. A vulnerability exists when the *QueryInterface* method fails and *ReleaseInterface* is still being called. This is due to (1) the *pUnk* pointer being uninitialized and (2) the return check being placed after the call to *ReleaseInterface*. The combination of those two bad coding practices leads to a security vulnerability due to calling method *Release* on an initialized interface pointer that will later trigger in the form of a use-after-free vulnerabilities. In order to exploit this flaw for untrusted code execution, an attacker would need to control the content of the stack and make the stack offset used by local variable *pUnk* coincides with the stack offset used by another reference-counted class in a previous stack context (for example, in a function previously called and already returned from, that held a local variable at the same stack offset just before the call to *CSomeElement::add* was performed). Such security exploits have already been demonstrated by industry researchers.

2) VARIANT type confusion: The second vulnerability class (Figure 12) that we present in this article is related to the VARIANT data structure. The VARIANT data structure is used in COM applications to transfer data items across generic interfaces. A generic container for arrays of VARIANT structures used in such COM programs is the *DISPPARAMS* structure. This container type is used, among others, by the *IDispatch::Invoke* interface method. A vulnerable specialization of this method is shown below. The *CBrowserOp* class derives from *IDispatch* and overloads its *Invoke* method. This derived method then assumes that the first VARIANT array element is an *IUnknown* interface pointer (which is the base interface for all COM classes and COM interfaces) and looks up the desired interface (using a call to *QueryInterface*) needed to perform the effective COM operation. The rest of the VARIANT array elements is then passed to this newly looked-up interface. On success, the action is performed and the interface is properly released.

The safety of VARIANT manipulation relies on testing its *vt* field to make sure that the contained pointer corresponds to a data type that the interface is expecting and able to treat. Failure to perform such check can be devastating for the security of code, especially if the input parameters are user-controlled. This method fails to perform such check before calling *QueryInterface* on the *IUnknown* pointer *pUnk*. If the first VARIANT structure field *dParams→rgvarg[0].pdispVal* were to contain another type of interface, a different method than *QueryInterface* would be called. An attacker could possibly redirect execution onto an instruction pointer of its choice, or, if the interface pointer is not directly controlled, on a different type of interface that implements a different set of methods, expecting different parameters, leading to a likely exploitable memory corruption in the program. The appropriate fix for this vulnerability is to extend the conditional predicate to insert a test `vt == VT_UNKNOWN || vt == VT_DISPATCH` to make sure that the VARIANT structure holds the appropriate interface pointer into which the *QueryInterface* method is implemented, and returns *E_INVALIDARG* if this is not the case.

VI. Related work

Extended static checking was pioneered for Java by the work of ESC/Java [FLL+02a] as a means to use program verification methods to provide high coverage of user defined assertions. It has since then been applied to other languages such as Spec# for C# [BLS05], and HAVOC for C [BHL+10]. Unlike these tools, HAVOC-LITE provides a rich set of instrumentation and inference capabilities to reduce the manual burden for large modules. HAVOC-LITE provides support for most common C++ features used in legacy applications.

Software model checkers such as SLAM [BMMR01] offer automatic annotation inference based on predicate abstraction for sparse type-state properties. However, these approaches are not known to scale to modules greater than 100KLOC, thus cannot be applied to most of the modules in this work. These tools do not provide mechanisms for users to configure the analysis by providing the set of candidates as HAVOC-LITE provides. Saturn [ABD+07] provides precise intraprocedural

```
1  STDMETHODIMP CBrowserOp::Invoke(DISPID dispId, DISPPARAMS *dParams)
2  {
3    switch (dispId) {
4      case DISPID_ONPROCESSINGCOMPLETE:
5        if (!dParams || !dParams->rgvarg || !dParams->rgvarg[0].pdispVal) {
6          return E_INVALIDARG;
7        }
8        else {
9         IUnknown *pUnk = dParams->rgvarg[0].pdispVal;
10        INeededInterface *pRes = NULL;
11        HRESULT hr = pUnk->QueryInterface(IID_NeededIface, (void **) &pRes);
12        if (hr == S_OK) {
13          PerformAction(pRes, dParams);
14          ReleaseInterface(pRes);
15        }
16       }
17      break;
18      default: return DISP_E_MEMBERNOTFOUND;
19   }
20   return S_OK;
21 }
```

Fig. 12. A real VARIANT type confusion vulnerability (obfuscated)

analysis using SAT solvers, and uses procedure summaries over a set of fixed vocabulary. The tool developer provides appropriate summaries for various properties such as memory leaks, and is not easily configurable. Saturn has been used to check specific security properties such as validation of user-land pointers passed to the kernel [BA08]; however, this required the careful configuration of the Saturn checker that cannot be expected of an average security expert.

Hackett et al. [HDWY06] provide a checker for buffer overruns in legacy applications using a combination of buffer length annotation and a set of custom rules to check these annotations. The checker uses a custom constraint solver (not modern SMT solvers) using a few simple rules. However, they provide useful heuristics to infer buffer annotations (relating buffers to their lengths) that significantly reduces the annotation effort. However, the technique cannot be readily extended to the properties we discuss in this work and therefore not a good match for security auditors.

Theorem provers have recently been quite successful in dynamic test generation tools such as DART [GKS05], EXE [CGP+06] and SAGE [GLM12]. These techniques leverage existing tests to create path constraints that can be negated to obtain tests for alternate paths. These techniques have revealed several bugs in large applications (in integration testing) or small libraries (in unit testing). However, these approaches are oblivious of the property being checked and aim at providing higher path coverage. These techniques do not use procedure summaries (such as those provided by our candidate annotations) and therefore cannot provide coverage guarantees on the entire attack surface. Since these techniques are primarily based on testing, they suffer from few false alarms. However, not all the crashes lead to security vulnerabilities and require a user to triage the bugs that can lead to exploits. On the other hand, several of the properties (e.g. double free) do not lead to crashes and are consequently harder to detect by testing based tools.

Security properties such as the *VARIANT* type consistency [DSD09] and the reference counting invariants [CCC10] have previously been studied for COM programs and web browsers via runtime monitoring and unit testing. However, no systematic program analysis was performed to the extent of this work. As such, no guarantee of coverage could be made based upon concrete executions of the program under scrutiny. HAVOC-LITE provides security auditors with extended security audit abilities and allow focusing on code locations where such security properties could not been proved. As such, extended static checking provides a much stronger guarantee that no instance of such vulnerability has been left behind.

Recent work on Automated Exploit Generation [ATD11] such as the one based on the Bitblaze framework [SBY+08] is attempting both vulnerability checking and input crafting based on the same toolset in order to find a code defect and force execution to be redirected on malicious code. Such approach has been focused on basic security properties such as buffer overflow induced by insecure API like *strcpy* . Such combination of SMT solvers and dynamic test generation is a promising direction provided the performance overhead is overcome and more properties are encoded for the checker. However, such framework provides very limited configurability for a security auditor and does not guarantee as much coverage as an extended static checker.

VII. Conclusions

Extended static checking is a good complement to fuzz testing and other data-flow based static analysis techniques. In particular, the inter-procedural inference is a key component in getting the signal/noise ratio to an acceptable level for security experts to review. While inference has shown improve the

signal/noise ratio substantially (sometimes up to 45%), the cost associated to running the inference is still high. Our goal is to extend the two-level Houdini algorithm to a distributed version that can invoke multiple versions of the inner Houdini algorithm in parallel. HAVOC-LITE currently does not weigh warning confidence and users have to go through the entire list of alerts to have a faithful understanding of the results — performing warning prioritization is an important next step for HAVOC-LITE.

REFERENCES

[ABD+07] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An overview of the Saturn project. In *Workshop on Program Analysis for Software Tools and Engineering (PASTE '07)*, pages 43–48, 2007.

[ATD11] Hao Brent Lim Tze Avgerinos Thanassis, Cha Sang Kil and Brumley David. Aeg: Automatic exploit generation. Network and Distributed System Security Symposium, 2011.

[BA08] S. Bugrara and A. Aiken. Verifying the safety of user pointer dereferences. In *IEEE Symposium on Security and Privacy*, pages 325–338, 2008.

[BHL+10] T. Ball, B. Hackett, S. K. Lahiri, S. Qadeer, and J. Vanegue. Towards scalable modular checking of user-defined properties. In *Verified Software: Theories, Tools, Experiments (VSTTE '10)*, volume LNCS 6217, pages 1–24, 2010.

[BL05] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Program Analysis For Software Tools and Engineering (PASTE '05)*, pages 82–87, 2005.

[BLS05] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS '05)*, pages 49–69, 2005.

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Programming Language Design and Implementation (PLDI '01)*, pages 203–213, 2001.

[CCC10] Shuo Chen, Hong Chen, and Manuel Caballero. Residue objects: a challenge to web browser security. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 279–292. ACM, 2010.

[CGP+06] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[Che02] B. Chess. Improving computer security using extended static checking. In *IEEE Symposium on Security and Privacy*, pages 160–, 2002.

[CHLQ09] J. Condit, B. Hackett, S. K. Lahiri, and S. Qadeer. Unifying type checking and property checking for low-level code. In *Principles of Programming Languages (POPL '09)*, pages 302–314, 2009.

[Com] Component Object Model (COM). http://www.microsoft.com/com/default.mspx.

[DL05] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[DMS+09] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. Vcc: Contract-based modular verification of concurrent c. In *International Conference on Software Engineering, (ICSE '09), Companion Volume*, pages 429–430, 2009.

[DSD09] M. Dowd, R. Smith, and David Dewey. Attacking interoperability. Blackhat USA briefings, 2009.

[FL01] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe (FME '01)*, pages 500–517, 2001.

[FLL+02a] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI'02)*, pages 234–245, 2002.

[FLL+02b] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Language Design and Implementation (PLDI'02)*, pages 234–245, 2002.

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Programming Language Design and Implementation (PLDI '05)*, pages 213–223. ACM, 2005.

[GLM12] P. Godefroid, M. Y. Levin, and D. A. Molnar. Sage: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.

[HDWY06] B. Hackett, M. Das, D. Wang, and Z. Yang. Modular checking for buffer overflows in the large. In *International Conference on Software Engineering (ICSE '06)*, pages 232–241, 2006.

[KAE+10] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

[LQR09] S. K. Lahiri, S. Qadeer, and Z. Rakamaric. Static and precise detection of concurrency errors in systems code using smt solvers. In *Computer Aided Verification (CAV '09)*, volume LNCS 5643, pages 509–524, 2009.

[Sat] Satisfiability Modulo Theories Library (SMT-LIB). Available at `http://goedel.cs.uiowa.edu/smtlib/`.

[SBY+08] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. Proceedings of the 4th International Conference on Information Systems Security, 2008.