Software Integrity Checking Expressions (ICEs) for Robust Tamper Detection

Mariusz Jakubowski[1,⋆] Prasad Naldurg[2], Vijay Patankar[2], and Ramarathnam Venkatesan[3]

[1] Microsoft Research Redmond
[2] Microsoft Research India
[3] Microsoft Research Redmond and Microsoft Research India
{mariuszj,prasadn,vij,venkie}@microsoft.com

Abstract. We introduce software integrity checking expressions (SoftICEs), which are program predicates that can be used in software tamper detection. We present two candidates, probabilistic verification conditions (PVCs) and Fourier-learning approximations (FLAs), which can be computed for certain classes of programs,. We show that these predicates hold for any valid execution of the program, and fail with some probability for any invalid execution (e.g., when the output value of one of the variables is tampered). PVCs work with straight-line integer programs that have operations $\{ *, +, - \}$. We also sketch how we can extend this class to include branches and loops. FLAs can work over programs with arbitrary operations, but have some limitations in terms of efficiency, code size, and ability to handle various classes of functions. We describe a few applications of this technique, such as program integrity checking, program or client identification, and tamper detection. As a generalization of oblivious hashing (OH), our approach resolves several troublesome issues that complicate practical application of OH towards tamper-resistance.

1 Introduction

We describe a general framework for generating and validating useful verification conditions of programs to protect against integrity attacks. We present two methods, one to generate probabilistic verification conditions (PVCs) and the other to compute Fourier-learning approximations (FLAs), which can both be viewed as instances of a general class of software integrity checking expressions (SoftICEs). These can be applied to a variety of problems, including software-client identification and tamper detection.

Our PVC technique relies on the transformation of straight line program fragments (without control-branching or loops) into a set of polynomial equations. With each set of equations, we compute a reduced basis that eliminates redundant variables and equations that do not contribute to the output. This basis

⋆ Corresponding author.

T. Furon et al. (Eds.): IH 2007, LNCS 4567, pp. 96–111, 2007.

depends on program code, and is consistent with the input-output semantics of a given program fragment. We show how we can use this basis as an integrity checking expression (ICE) with provable security properties.

Similary, in our FLA method, we view a code fragment as a function operating on variables read inside the fragment; the output is all variables potentially overwritten within the fragment. This scheme treats such a function as a series of component functions that map the input variables onto single bits. A Fourier-based machine-learning technique converts such functions into tables of Fourier coefficients, and an inverse transform can use this table to approximate the original function. Together with the coefficients, the inverse transform serves as an ICE to verify each individual bit of the target function.

Traditionally, verification conditions (*VC*s) are used formally (or axiomatically) to validate properties of programs without actually executing all possible paths in the program. In particular, these conditions characterize a computable semantic interpretation, which is a mathematical or logical description of the possible behaviors of a given program [1]. In this context, for example in [2,3], one typically has a specification of the property of interest, and the generation of VCs is driven by this property. These techniques typically work on a formal model that is an over-approximation or abstraction of program behavior. One of the challenges in abstraction is this loss of precision, and leads to false-errors. On the other hand, we are interested in capturing verification conditions that are precise, and characterize input-output behavior accurately.

In addition to this, we differ from traditional VC generation in many aspects: We are agnostic to particular property specifications and therefore completely automatic. Recently, a technique called random interpretation has been proposed [6], which combines abstract interpretation with random testing to assert probabilistic property-driven conditions for linear programs. Our framework describes how to generate generic semantic fingerprints of programs, independent of property specifications or particular verification frameworks (e.g., abstract interpretation). Furthermore, most existing techniques for generating verification conditions work only for linear constraints. Our methods are more general and work with nonlinear integer programs as well.

Our work on PVCs is also related to previous work on fast probabilistic verification of polynomial identities [4], but our polynomials are derived from actual programs.

We envision the application of these conditions in two scenarios: (a) In a setting when the program is *oblivious* to the checks being performed, and (b) in a non-oblivious setting, where an adversary can try to learn the checks, which reduces to finding the random primes that are chosen for these reductions and creation of VCs. Our analysis is universal in the sense that we do not assume anything about the inputs.

The techniques proposed in this paper have a number of interesting applications:

- One application of our technique is *program identification* and individualization. Consider a web portal W that distributes client software that is

individualized with respect to some client identity information. Now, imagine some other portal W' that wants to use the database and services of W to re-package and sell the same software to its clients. If W wants to keep track of its original code, it can embed our ICEs in it.

– Another application is *software protection*. We believe that our probabilistic conditions can be used as primitives to build provably secure software protection mechanisms. Desirable properties of software protection include obfuscation and tamper-resistance. Security is determined by the minimum effort required to bypass such measures, and provable security means that we can accurately estimate this effort, even if it is mere hours or days.

The problem of tamper detection in an oblivious setting has been explored in [10]. This work proposes a technique called *oblivious hashing (OH)*, which computes hash values based on assignments and branches executed by a program. OH requires program inputs that exercise all code paths of interest; hashes may then be pre-computed by executing the program on these inputs. Alternately, OH may rely on code replicas to compute and compare redundant hashes. In contrast, we require no specific inputs, hash pre-computation, or code replicas, thus resolving a number of issues that have created roadblocks against practical adoption of OH.

A number of recent results [7,8,9] have explored the nature and scope of program obfuscation. These results have identified different classes of programs that can (or cannot) be obfuscated, for a general definition of obfuscation as being equivalent to black-box access to a program. So far these results have only shown very small classes of programs that can be provably obfuscated. The properties defined here are not quantitative and we believe we can extend the scope of these results to apply to a broader class of problems with probabilistic guarantees.

In a non-oblivious setting, the notion of proof-carrying code was developed in [5], where an untrusted program carries a proof for a property defined by the verifier. The verifier generates VCs automatically, using the same algorithm as the program developer, and checks the proof. If the proof cannot be verified, the program is considered unauthentic. In contrast, our probabilistic predicates assert invariants about input-output behavior, and we believe that they can be used to detect tampering and aid in obfuscation.

The rest of the paper is organized as follows: We present our program model in Section 2. Section 3 presents the main theorem, about the precision of these predicates, as well as their failure-independence when we trade precision for efficiency. Section 4 presents examples, including extensions to include branches and loops, and comparisons with traditional VC-generation techniques. Section 5 concludes with a discussion on future work.

2 Program Model and Basics

In this section, we describe our program model, along with our assumptions, and present mathematical preliminaries, including definitions and notations, that are

needed to explain our PVC technique. We explain our FLA technique, and its associated model in Section 5.

We study this problem in the context of integer programs. Here we allow only integer values to variables in all (possibly infinite) reachable program-states, consistent with our program semantics. In particular, we restrict our variables to take values in the ring of rational integers (denoted by $\mathbb{Z}$). Here, we would like to point out that our method can be easily adapted to deal with programs with rational number inputs, i.e. inputs from $\mathbb{Q}$. This is achieved by considering a rational number input as a quotient of two integers. Thus, each variable (with rational number input) in an assignment is replaced by two distinct variables (quotient of two variables) that accept integer inputs. One can then *homogenise* the resulting assignment and produce an assignement with integer inputs. Repeated application of this process will covert a given program with rational number inputs into an equivalent program with integer inputs. Thus, without loss of generality, we may restrict our study to programs with integer inputs.

In our PVC technique we exploit the fundamental correspondence between ideals of polynomial rings and subsets of affine spaces, called affine varieties, to generate our probabilistic conditions.

Our goal is to capture a precise algebraic description of the relationship between input and output variables for a generic program. To this end, with each program P we associate an ideal or equivalently a system of polynomial equations $f_i(x_1, \cdots, x_n) = 0$. We then compute a basis of such an ideal by eliminating variables and redundant equations. This basis has the same set of zeros as the original program. In Section 3, we show how we can construct a probabilistic predicate from this basis and describe a testing framework to detect tampering.

2.1 Background and Notation

To make this paper self-contained, we present definitions of rings, polynomial rings, ideals and their bases. Our domain of program variables will be $\mathbb{Z}$, the ring of rational integers.

A *ring* R is a set equipped with two operations, that of multiplication and addition $\{+, .\}$, together with respective *identity* elements denoted by 1 and 0. A ring is *commutative* if the operation of multiplication is commutative. Henceforth we will only deal with commutative rings.

An *ideal* of a ring R is an additive subgroup of R. In other words, an ideal I of a commutative ring R is a nonempty subset R such that $(I, +)$ is a subgroup of $(R, +)$ and that for all $r \in R$ and $x \in I$, $r \cdot x \in I$. An ideal *generated* by a given subset S of R is by definition the smallest ideal of R containing S. This ideal is denoted by $\langle S \rangle$ and called as the ideal of R generated by S.

Let $\{x_1, x_2, \cdots, x_n\}$ be n indeterminate *algebraically* independent variables over a commutative ring R, where n is a positive integer. Let $R[x_1, x_2, \cdots, x_n]$ be the ring of polynomials over R. We will denote this for short by $R[\bar{x}]$. Note that, as said earlier, it is a ring under the operations of multiplication and addition of polynomials. We say that a commutative ring R is a field if every non-zero element is invertible or has an inverse. A field will be usually denoted by K or

k. Thus, let k be a field, and let $k[\bar{x}]$ denote the polynomial ring over k. If an ideal I of R is such that there exists a finite subset $X \subseteq R$(necessarily a subset of I) generating it, then the ideal I is said to be finitely generated. It is a basic theorem that every ideal of $\mathbb{Z}[x_1, x_2, \cdots, x_n]$, the polynomial ring of integers, is finitely generated.

We now focus on *Gröbner bases*, a particular type of generating subset of an ideal in a polynomial ring. It is defined with respect to a particular monomial ordering, say $\preceq$. By a monomial orderning, we mean a way of comparing two different monomials in n variables over R. It is a theorem that every ideal posseses a unique Gröbner basis depending only on the monomial ordering $\preceq$. Thus for a fixed monomial ordering $\preceq$, we will denote the Gröbner basis of an ideal I by G. Thus, we can write $G := \{g_1, \cdots, g_m\}$, for some polynomials g_i, and $< G >= I$.

Gröbner bases possess a number of useful properties. The original ideal and its Gröbner basis have the same zeros. The computation of a Gröbner basis may require time (in the worst case) that is exponential or even doubly-exponential (for different orderings) in the number of solutions of the underlying polynomial system (or ideals). However, we have observed that they are efficiently computable in practice, in a few seconds, for typical code fragments of interest, and most computer algebra packages such as Mathematica and MAGMA provide this support. We propose to validate their usefulness for checking program fragments related to license-checking and digital rights protection. Computation of our ICEs is off-line, in the sense that it can be viewed as a precomputation and this stage does not affect the runtime performance of our original applications.

Reduced Gröbner bases can be shown to be unique for any given ideal and monomial ordering. Thus, one can determine if two ideals are equal by looking at their reduced Gröbner bases.

Next, we show how one may use these properties to generate behavioral fingerprints of program executions.

2.2 Computing Bases

We explain our technique for straight-line programs in this subsection. We focus on program fragments that form a part of what is called a basic-block, without any additional control flow instructions. Subsquently, we present some engineering techniques with weaker guarantees that can handle control flow branching and looping. However, this section is of independent value as we can apply our general technique to only program fragments that are sensitive, trading coverage for performance.

Let P be a straight-line integer program fragment. Let $x_1, \cdots, x_r$ be all the input variables, and let $x_{r+1}, \cdots, x_n$ be all the output variables of P. We define the set of program states $V(P)$, as the set (possibly infinite) of all possible valuations to the variables $x_1, \cdots, x_n$ of P, consistent with the update semantics of variables in the program.

We assume that the *operations* of P are defined over the the integers $\mathbb{Z}$ and are restricted to addition, subtraction, and multiplication by integers and combinations of quantities obtained by these.

As mentioned earlier, with homogenisation and other algebraic simplifications, our method can be easily adapted for programs that include the division operation. We can convert a given assignment that contains quotients of polynomials into new assigment (or assignments) without quotients (or division operation), by introducing auxiliary variables when necessary.

Therefore, we view a straight-line program (without any branches and conditions) as a set of polynomial equations in some finite variables with integer coefficients. In order to view assignments as equations, we use a standard transformation technique called *Static Single Assignment (SSA)*, which converts an ordered sequence of program statements into a set of polynomials by introducing temporary variables. In SSA, if a program variable x is updated, each new assignment of x is replaced with a new variable in all expressions between the current assignment and the next. One thus gets a set of polynomial equations associated with a given program. Let $I(P)$ be the ideal generated by these polynomials. This ideal will be called as the *Program Ideal* of the given program P. Now, if we fix a monomial ordering of the variables that are involved in the definitions of $I(P)$, we can construct a Gröbner basis for $I(P)$. Let us denote it by $G(P)$. This gives us the following: The set of states that evaluate to zero for a Gröbner basis is identical to the set of states that evaluate to zero for the original polynomials. In this sense, as a VC, the Gröbner basis as an abstraction of the program behavior is **precise**.

This can be utilised as follows: Suppose $x_i = \lambda_i$ for $i = 1$ to n is a specific executable-instance of a program P. Then, $g_j(\bar{\lambda}) = 0$ for $j = 1$ to m, here $\bar{\lambda} := (\lambda_1, \cdots, \lambda_n)$. Thus, if we take up all the polynomials of G, and evaluate $\bar{\lambda}$ at these, then we can verify *authenticity* of program P with respect to its input-output behaviour by checking whether an execution-instance of P is satisfied by all the polynomials of G. However, this would constitute a lot of checking.

Rather than check for authenticity, it is easier to check for tampering. If a program is modified or tampered and its input-output behavior has changed, the bases produced by the original program and the modified program will be different. For a given set of inputs, if the program is not tampered, the Gröbner bases associated with this will evaluate to zero. However, if they evaluate to non-zero, then the two programs are not the same. If we can find an input-output for which the Gröbner Basis is non-zero for a tampered program, we can assert (by black-box testing) that the program has been tampered. Finding this particular input instance that will cause the basis to evaluate to non-zero is also difficult. However, with the reduction presented in the next section, we show how we can rely on a number-theoretic argument to quantify the security of our scheme for a general adversary without having to rely on particular input instances.

In the next section, we present our main theorem regarding probabilistic generation and validation of predicates using our basic idea.

3 Probabilistic Verification Conditions

In this section, we derive a probabilistic validation property that is independent of specific program inputs and outputs. We use a simple number-theoretic method of reduction modulo primes.

We now study how the Gröbner Basis polynomials behave when we apply reduction modulo primes. This is useful for a variety of reasons. If we can quantify how often the reduced polynomials produce the same set of zeros in comparison with the original polynomials, we can devise a probabilistic testing framework that complicates the task of a tampering adversary.

We employ the Schwartz-Zippel lemma [4] used in standard testing of polynomial identities to obtain this quantification, which is typically used to determine if a given multivariate polynomial is equal to zero.

Theorem 1 (Schwartz-Zippel). *Let $P \in F[x_1, x_2, \cdots, x_n]$ be a (non-zero) polynomial of total-degree $d > 0$ over a field F. Let S be a finite subset of F. Let $r_1, r_2, r_3, \cdots, r_n$ be selected randomly from S. Then*

$$Pr[P(r_1, r_2, \cdots, r_n) = 0] \leq \frac{d}{|S|}.$$

This is basically a generalised version of the fact that a one variable polynomial of degree d has at most d roots over a field F.

We will use a variation of the above lemma, which we state below. This variation follows from the earlier lemma by noting that a random choice of $r \in \mathbb{Z}$ amounts to a random choice of $r \mod p$ in $\mathbb{Z}/p\mathbb{Z}$ for a randomly chosen prime p and that $|S| = |\mathbb{Z}/p\mathbb{Z}| = p$.

Theorem 2 (Schwartz-Zippel-Variant). *Let $P \in Z[x_1, x_2, \cdots, x_n]$ be a (non-zero) polynomial of total-degree $d > 0$ defined over the integers $\mathbb{Z}$. Let P be the set of all primes numbers. Let $r_1, r_2, r_3, \cdots, r_n$ be selected randomly from $\mathbb{Z}$. Then*

$$Pr[P(r_1, r_2, \cdots, r_n) = 0 \mod p\ ] \leq \frac{d}{p}.$$

Thus, the Schwartz-Zippel lemma bounds the probability that a non-zero polynomial will have roots at randomly selected test points. If we choose a prime $p > d$, given a polynomial from a Gröbner basis that is computed for a straight-line program as described earlier, the probability that this polynomial will be zero when evaluated at a random input-output (of the given program) is bounded above $\frac{d}{p}$.

This quantification provides us a basis for defining a probabilistic testing methodology as follows:

If we are given black-box access to tampered code, the probability of the code producing a *zero* (an input-output instance at which the polynomial evaluates to zero) will be bounded by $\frac{d}{p}$. But p is chosen at random and can be arbitrarily

large. Thus, in order to pass tampered code as authentic, an adversary will have to guess a random prime from a possibly infinite set of choices, and this is a well-known hard problem.

Furthermore, granted that the adversary knows the prime, the adversary also needs to maximize the probability that a non-authentic input-output instance is passed on as authentic. Equivalently, the adversary needs to make sure that a random and tampered input-output instance be a zero of all the polynomials (of a Gröbner bases) modulo p. Given p and a finite set of polynomial equations, this can be achieved with some work, provided the polynomial equations are *simple*. Instead if randomized techniques are employed, and if the prime p is much larger than the total-degree of all the polynomial equations, then the difficulty of finding a zero has already been quantified by the Schwartz-Zippel lemma.

On the other hand, the verifier can test the program for random input-outputs, and modulo a randomly chosen large prime p. If the program is not tampered, all the input-outputs will be the zeros for the polynomials modulo any prime p. The more tests the verifier does, the lesser the error probability. But, by the arguments above, the probability that a tampered input-output instance passes as a zero of a polynomial modulo a random large prime p is bounded above by $\frac{d}{p}$. The probability of passing a specific non-authentic instance as authentic can be minimized by choosing many randomly chosen primes p_i and repeating the verification on the same given specific instance as needed.

4 Examples of PVCs

In this section, we present five examples to demonstrate how our technique can be used in practice. The first example shows how we can generate conditions for linear programs. We also show how our probabilistic conditions compare with traditional verification conditions. The second example has non-linear constraints. The third example presents is more exploratory and presents some preliminary ideas on branches and loops. In the fourth example, we show how we can compute these bases for small, overlapping, randomly selected code fragments to scale our solution to large programs. Finally we show how we can reduce the complexity of checking by using our results from our previous section.

4.1 Linear Programs

In the following example, the input and output variables in this progr are $\{x, y, z\}$.

$$
\begin{aligned}
x &= x + y + z;\\
y &= y + 5;\\
z &= x + 1;\\
x &= x + 1;
\end{aligned}
$$

In order to treat these assignment as equations, we transform the program using SSA into the following:

$$\begin{aligned} x_1 &= x_0 + y_0; \\ x_2 &= x_1 + z_0; \\ y_1 &= y_0 + 5; \\ z_1 &= x_2 + 1; \\ x_3 &= x_2 + 1; \end{aligned}$$

In the example above:

$$\begin{aligned} I = \langle & x_1 - x_0 - y_0, x_2 - x_1 - z_0, \\ & y_1 - y_0 - 5, z_1 - x_2 - 1, x_3 - x_2 - 1 \end{aligned}$$

The Gröbner basis of this ideal with respect to a fixed monomial order $\{x_0 < x_1 < x_2 < x_3 < y_0 < y_1 < z_1\}$ is given by:

$$\begin{aligned} G = \{ & 5 + y_0 - y_1, x_3 - z_1, 1 + x_2 - z_1, \\ & 1 + x_1 + z_0 - z_1, -4 + x_0 + y_1 + z_0 - z_1\} \end{aligned}$$

When the order is changed we get a different basis. For the ordering $z_0 < y_0 < y_1 < x_0 < x_1, x_2 < x_3$ the basis is $\{x_3 - z_1, 1 + x_2 - z_1, -5 + x_0 - x_1 + y_1, x_0 - x_1 + y_0, 1 + x_1 + z_0 - z_1\}$ For both these cases, the basis polynomials evaluate to zero for any valuations to input variable x_0, y_0, and z_0. However if the program output is changed (simulated by changing some intermediate outputs, these polynomials do not evaluate to zero.

Comparison with Traditional VC Generation. For comparison, we compute invariants using a standard strongest-precondition algorithm. Suppose we start with an assumption $x > 0, y > 0, z > 0$. The VC obtained in this case is $z_1 \geq 4 \wedge x_3 \geq 4 \wedge y_1 \geq 6$. Note that if we use these assertions in a black-box or oblivious setting, we can argue trivially that they are less resilient to program modification than the ones generated by our technique.

If we start with true as the initial assertion, i.e., $\top$, we get:

$$\begin{aligned} VC = \langle & (z_1 = x_0 + y_0 + z_0 + 1) \wedge \\ & (x_3 = x_0 + y_0 + z_0 + 1) \wedge \\ & (y_1 = y_0 + 5) \rangle \end{aligned}$$

While the strongest-postcondition algorithm now produces an equivalent set of conditions (and depended on what we gave to it initially), our technique can produce probabilistic conditions, and can be applied to nonlinear programs as well.

4.2 An Automated Nonlinear Example

Below we give an example of Gröbner-basis computation on typical code in an automated fashion. For this, we have implemented an SSA-remapping tool that converts C++ code into polynomials suitable for our basis computation. Consider the following code snippet:

$$
\begin{aligned}
x &= b^2 + 2a - 17c;\\
y &= x + 3ab;\\
z &= 19b - 18yx^2;\\
y &= x + 2y - z;
\end{aligned}
$$

After processing the above input code, our tool generates the following polynomials:

```
t154 - (b0 * b0),          y0 - t161,
t155 - (2 * a0),           t162 - (19 * b0),
t156 - (t154 + t155),      t163 - (18 * y0),
t157 - (17 * c0),          t164 - (t163 * x0),
t158 - (t156 - t157),      t165 - (t164 * x0),
x0 - t158,                 t166 - (t162 - t165),
t159 - (3 * a0),           z0 - t166,
t160 - (t159 * b0),        t167 - (2 * y0),
t161 - (x0 + t160),        t168 - (x0 + t167),
t169 - (t168 - z0),        y1 - t169
```

In the above, variables with names prefixed by 't' (e.g., `t154`) are new temporaries introduced by our SSA-remapping tool. Original variables (e.g., `y`) are extended with numerical suffixes to create SSA-remapped versions (e.g., `y0,y1`). Only the variable `y` requires more than one version, since only `y` is assigned more than once.

With variables `t154` through `t169` eliminated, a Gröbner basis for the above polynomials is the following:

$$
\begin{aligned}
&x_0 + 2y_0 - y_1 - z_0,\\
&2a_0 + {b_0}^2 - 17c_0 + 2y_0 - y_1 - z_0,\\
&3a_0b_0 - 3y_0 + y_1 + z_0,\\
&6{a_0}^2 - 51a_0c_0 + 6a_0y_0 + 3b_0y_0 - 3a_0y_1 -\\
&b_0y_1 - 3a_0z_0 - b_0z_0,\\
&-19b_0 + 72{y_0}^3 - 72{y_0}^2y_1 + 18y_0{y_1}^2 + z_0 -\\
&72{y_0}^2z_0 + 36y_0y_1z_0 + 18y_0{z_0}^2
\end{aligned}
$$

The above basis polynomials evaluate to zero on any set of proper assignments to the variables $a_0, b_0, c_0, x_0, y_0, z_0$, and y_1. For example, if $a_0 = 3$ and $b_0 = 14$

and $c_0 = 15$, we have $x_0 = -53$, $y_0 = 73$, $z_0 = -3690760$, $y_1 = 3690853$, and each basis polynomial evaluates to zero on these assignments. If an attack or a program error tampers with these values, this will most likely no longer hold. For example, if the the value of y_0 is changed from 73 to 72, the five basis polynomials evaluate to $\{-2, -2, 3, -60, -320130\}$.

Note that the variable y_0 corresponds to the value of `y` prior to its second assignment in the original C++ snippet; y_1 is the final value of `y` computed by the C++ code.

4.3 Conditionals and Loops

To generate VCs for conditional statements, we compute VC sets independently for each branch path; we then perform a cross product of these VC sets. Since all polynomials in at least one VC set must evaluate to zero, each polynomial in the cross product must also vanish. As an example, consider the following C++ snippet:

```
if (...)                    else
{                           {
  x = b*b - 17*a*b;         x = b*b - 2*a + 17*c;
  x = x - 3*x*c;            y = x + 2*a*b;
}                           }
```

Polynomials corresponding to the two branch paths are as follows:

$$x_0 - b^2 + 17ab,$$
$$x_1 - x_0 + 3x_0c$$

$$x_0 - b^2 + 2a - 17c,$$
$$y - x_0 - 2ab$$

The respective Gröbner bases are:

$$-x_0 + 3cx_0 + x_1,$$
$$17ab - b^2 + x_0$$

$$-2a + b^2 + 17c - x_0,$$
$$2ab + x_0 - y,$$
$$4a^2 - 34ac + 2ax_0 + bx_0 - by$$

The cross product of these bases consists of 6 polynomials, each of which must evaluate to zero on any proper variable assignment:

$$
\begin{aligned}
&(-x_0 + 3cx_0 + x_1)(-2a + b^2 + 17c - x_0),\\
&(-x_0 + 3cx_0 + x_1)(2ab + x_0 - y),\\
&(-x_0 + 3cx_0 + x_1)(4a^2 - 34ac + 2ax_0 + bx_0 - by),\\
&(17ab - b^2 + x_0)(-2a + b^2 + 17c - x_0),\\
&(17ab - b^2 + x_0)(2ab + x_0 - y),\\
&(17ab - b^2 + x_0)(4a^2 - 34ac + 2ax_0 + bx_0 - by)
\end{aligned}
$$

Note that this method produces VCs that ascertain the proper execution of each branch path; however, these VCs do not verify that the proper path was chosen according to the condition evaluated at runtime. To fix this, the condition itself may be treated as a polynomial for VC generation. Future work will include details of how this may be accomplished.

To handle a loop, we may compute a Gröbner basis for just the loop body. While this will not yield VCs that verify the actual loop iteration, we may additionally include loop variables and conditions in the set of input polynomials. Alternately, we may unroll loops, producing new instances of loop variables for each iteration. A more detailed description of these methods will appear in future work.

4.4 Overlapping

For larger code sections, computing Gröbner bases may be expensive. Even with modular reduction, the results may contain an unwieldy number of complex polynomials. Moreover, depending on the order of monomial elimination, the time and resources to compute a basis for large code sections may vary dramatically. In practice, well optimized software implementations are able to compute Gröbner bases for up to a few tens of variables.

To address this problem, we compute Gröbner bases for small, randomly overlapping fragments of input code; we then use a combination of the resulting VCs. This reduces the number and complexity of basis polynomials while retaining soundness and security. In addition, the overlapping creates links among the small code fragments, resulting in VCs that provide a probabilistic degree of precision.

As an example, consider the following C++ code segment:

```
x = b*b + 2*a - 17*c;
y = x + 3*a*b;
z = 19*b - 18*y*x*x;
y = x + 2*y - z;
```

We may split this into the overlapping fragments below, computing separate Gröbner bases for each:

```
x = b*b + 2*a - 17*c;
y = x + 3*a*b;
z = 19*b - 18*y*x*x;

y = x + 3*a*b;
z = 19*b - 18*y*x*x;
y = x + 2*y - z;
```

In general, the combined Gröbner bases from all fragments should be less complex and more usable than the single basis computed from the entire code segment. In future work, we will analyze the benefits and limitations of overlapping for purposes such as program analysis and tamper-resistance.

4.5 Reducing Complexity

We show a crucial number-theoretic trick with known bounds to reduce complexity and simplify analysis. We also emphasize the probabilistic nature of our conditions and highlight that we can analyze program behavior without making any assumptions on input-output models.

Consider a transformed program consisting of the following polynomials:

$$\begin{aligned} Q = \{ & x_1 - 2a + b + c, \\ & x_2 - 17a + b - 7c - 10, \\ & x_3 - 5b + a + 2, \\ & x_4 + 18a - 7b + c - 14\} \end{aligned}$$

The Gröbner Basis with $\{a, b, c\}$ eliminated is $-3154 + 497x_1 + 92x_2 - 88x_3 + 147x_4$. This evaluates to zero for any assignments to input variables. We now study how the polynomials reduce modulo a prime.

$$\begin{aligned} p = 2 &: \{x1 + x_4\} \\ p = 3 &: \{2 + 2x1 + 2x2 + 2x3\} \\ p = 7 &: \{3 + x2 + 3x3\} \\ p = 11 &: \{3 + 2x1 + 4x2 + 4x4\} \\ p = 17 &: \{8 + 4x1 + 7x2 + 14x3 + 11x4\} \\ p = 23 &: \{20 + 14x1 + 4x3 + 9x4\} \\ p = 101 &: \{78 + 93x1 + 92x2 + 13x3 + 46x4\} \end{aligned}$$

When we modify the outputs slightly, the bases now only evaluate to zero every $(1/p)$ times on an average, as expected. For example, with $p = 2$:

$$\begin{aligned} &\{0\}\{1\}\{0\}\{1\}\{0\}\{1\}\{0\}\{1\}\{0\}\{1\} \\ &\{0\}\{1\}\{0\}\{1\}\{0\}\{1\}\{0\}\{1\}\{0\}\{1\} \\ &\{0\}\{1\}\{0\}\{1\}\{0\}\cdots \end{aligned}$$

With $p = 11$:

$$\{0\}\{2\}\{4\}\{6\}\{8\}\{10\}\{1\}\{3\}\{5\}\{7\}$$
$$\{9\}\{0\}\{2\}\{4\}\{6\}\{8\}\{10\}\{1\}\{3\}\{5\}$$
$$\{7\}\{9\}\{0\}\{2\}\{4\}\cdots$$

With $p = 101$:

$$\{13\}\{40\}\{67\}\{94\}\{20\}\{47\}\{74\}\{0\}\{27\}$$
$$\{54\}\{81\}\{7\}\{34\}\{61\}\{88\}\{14\}\{41\}\{68\}$$
$$\{95\}\{21\}\{48\}\{75\}\{1\}\{28\}\{55\}\cdots$$

Subsequently, we hope to relax our restriction that the operations in our program be algebraic.

5 ICEs Via Fourier Learning

In this section, we show how to use a technique from machine learning to generate ICEs for arbitrary code, including mathematical operations and control-flow constructs. Our main idea is to treat a program fragment F as a function $f : \{0,1\}^n \to \{0,1\}^m$, and learn each such function via a standard training algorithm, as described shortly. The input to f is all variables read in F; the output of f is all variables potentially overwritten by F. For simplicity, assume that F modifies only a single bit and rewrite f as $f : \{0,1\}^n \to \{+1,-1\}$. For program fragments that change m bits, our scheme may consider m separate functions, one for each bit.

A Fourier-based learning procedure [13] essentially performs an approximate Fourier transform of a Boolean function $f : \{0,1\}^n \to \{+1,-1\}$. A basis for such functions is the family of functions $\chi_\alpha : \{0,1\}^n \to \{+1,-1\}$ defined as

$$\chi_\alpha(x) = (-1)^{\sum_{i=1}^{n} x_i \alpha_i}, \tag{1}$$

where α is an n-bit parameter, while x_i and α_i represent individual bits of x and α, respectively, for $i = 1..n$. Informally, each basis function $\chi_\alpha(x)$ computes the parity of a subset of x's bits, with the subset specified by the bit vector α. It can be shown that this function family is an orthonormal basis, so that a standard Fourier transform can map f onto 2^n Fourier coefficients c_α. These coefficients can be used to compute f:

$$f(x) = \sum_{\alpha \in \{0,1\}^n} c_\alpha \chi_\alpha(x) \tag{2}$$

With this basis, a full Fourier transform requires exponential time to compute an exponential number of coefficients. However, a typical function f can often be approximated by a small subset thereof. Efficient learning algorithms exist for functions that can be well approximated by a small number of coefficients

or by a selection of "low-frequency" coefficients c_α, where α has low Hamming weight. These algorithms use sets of "training" inputs and outputs to estimate values of coefficients, essentially approximating a Fourier transform.

Within this framework, an ICE for a program fragment F is a table of learned Fourier coefficients c_α for an associated bit function f, along with the expression specified by eq. 2. The coefficients and eq. 2 can be used to approximate $f(x)$ on any input x. As with any ICE, this result should match the actual value of $f(x)$ computed at runtime; otherwise F has been tampered.

As an example, we applied a "low-frequency" learning algorithm [13] to the following C fragment, which accepts the 12-bit integer variable x as input and returns a single-bit output:

```
uint y;
if ((x & 1) == 0)
  y = x >> 3;
else
  y = x >> ((int)x & 7);

if ((x & 2) == 0)
  y = 19 * y + x;
else
  y = y - 3 * x;

if ((x & 4) == 0)
  y = y * 11 + 1;
else
 y = 3 * y - 45;

return (0 == (y & 4)) ? 1 : -1;
```

To illustrate some specific figures, our learning procedure used 55690 random input-output pairs to approximate 1585 low-frequency Fourier coefficients c_α (with α having Hamming weight of 5 or less). This was sufficient to learn the above function well; e.g., for several random inputs x, the following lists the correct and approximated outputs:

```
x=372:  y=-1    y_approx=-1.54975758664033
x=648:  y=1     y_approx=0.855773029269167
x=3321: y=1     y_approx=1.09868917220327
x=1880: y=-1    y_approx=-0.807793140599749
```

This approach also provides other benefits for software protection, mainly due to obfuscation via homogenization. Programs turn into tables of Fourier coefficients, while execution becomes evaluation of inverse Fourier transforms (eq. 2). Thus, both representation and operation of transformed programs are highly uniform, which complicates analysis and reverse engineering. This is similar to representing functions as lookup tables, but Fourier-based learning works even when the size of such tables would be impractical. An adversary may be forced to treat Fourier-converted programs as black-box functions, since it is unclear how to recover original code from corresponding tables of coefficients. Future work will analyze the exact difficulty of this problem.

6 Future Work

This work should be considered as a preliminary investigation in our quest for robust, automatic, provably secure semantic fingerprints of programs. In future work, we will address computation of VCs for loops (with algebraic loop-variable updates) and conditionals. Moving beyond algebraic restrictions to accommodate bitwise and other operations, we may encode these computations as boolean formulas and arithmetize them suitably to treat them as algebraic polynomials (e.g., [11]). While this may increase the number of variables dramatically, we will use overlapping and other engineering techniques to manage this complexity for target applications.

References

1. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fix points. In: 4th Annual ACM Symposium on Principles of Programming Languages, pp. 234–252 (1977)
2. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic Predicate Abstraction of C Programs. PLDI 2001, SIGPLAN Notices 36(5), 203–213 (2001)
3. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with Blast. In: Ball, T., Rajamani, S.K. (eds.) Model Checking Software. LNCS, vol. 2648, pp. 235–239. Springer, Heidelberg (2003)
4. Schwartz, J.T.: Fast probabilistic algorithms for verification of polynomial identities. JACM 27(4), 701–717 (1980)
5. Necula, G.C.: Proof Carrying Code. In: 24th Annual ACM Symposium on Principles of Programming Languages, ACM Press, New York (1997)
6. Gulwani, S., Necula, G.C.: Discovering affine equalities using random interpretation. In: 30th Annual ACM Symposium on Principles of Programming Languages, pp. 74–84 (January 2003)
7. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (Im)possibility of Obfuscating Programs. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, Springer, Heidelberg (2001)
8. Kalai, Y.T., Goldwasser, S.: On the Impossibility of Obfuscation with Auxiliary Inputs. In: Proc. 46th IEEE Symposium on Foundations of Computer Science (FOCS 2005) (2005)
9. Lynn, B., Prabhakaran, M., Sahai, A.: Positive Results and Techniques for Obfuscation. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, Springer, Heidelberg (2004)
10. Chen, Y., Venkatesan, R., Cary, M., Pang, R., Sinha, S., Jakubowski, M.: Oblivious hashing: a stealthy software integrity verification primitive. In: Proceedings of the 5th International Workshop on Information Hiding, pp. 400–414 (2002)
11. Shamir, A.: IP = PSPACE. Journal of the ACM 39(4), 869–877 (1992)
12. Jacobson, N.: Basic Algebra I. W H Freeman and Co., New York (1985)
13. Mansour, Y.: Learning boolean functions via the Fourier transform. In: Roychowdhury, V., Siu, K.-Y., Orlitsky, A. (eds.) Theoretical Advances in Neural Computation and Learning, Kluwer Academic Publishers, Dordrecht (1994)