Fast Flow Analysis with Gödel Hashes

Shuying Liang
University of Utah
liangsy@cs.utah.edu

Weibin Sun
University of Utah
wbsun@cs.utah.edu

Matthew Might
University of Utah
might@cs.utah.edu

***Abstract*—Flow analysis, such as control-flow, data-flow, and exception-flow analysis, usually depends on relational operations on flow sets. Unfortunately, set related operations, such as inclusion and equality, are usually very expensive. They can easily take more than 97% of the total analyzing time, even in a very simple analysis. We attack this performance bottleneck by proposing Gödel hashes to enable fast and precise flow analysis. Gödel hashes is an ultra compact, partial-order-preserving, fast and perfect hashing mechanism, inspired by the proofs of Gödel's incompleteness theorems. Compared with array-, tree-, traditional hash-, and bit vector-backed set implementations, we find Gödel hashes to be tens or even hundreds of times faster for performance in the critical operations of inclusion and equality. We apply Gödel hashes in real-world analysis for object-oriented programs. The instrumented analysis is tens of times faster than the one with original data structures on DaCapo benchmarks.**

I. Introduction

Flow analysis, such as control-flow analysis, data-flow analysis and exception-flow analysis, usually depends on relational operations on flow sets (e.g. higher-order flow sets, points-to sets, *etc.*). Unfortunately, set related relational operations, such as inclusion and equality, are usually very expensive. In flow analysis, they can easily become the major performance bottleneck.

To expose this problem, we take a typical control flow analysis, the path-sensitive 0CFA as an example. Path-sensitive 0CFA is deeply exponential, and its theoretical utility lies in generating transition graphs that can be model-checked for safety and liveness properties [17]. The path-sensitive version of 0CFA [24], [23], [16] could perform on the order of $O(n^2((2^n)^n)^2)$ subsumption tests: programs as small as 100 lines can easily visit well over a hundred thousand states, requiring over a billion subsumption tests between *abstract heaps*.

To further show the performance bottleneck caused by set operations in a real analysis, we have instrumented a publicly available program analyzer that is developed for DARPA's Automated Program Analysis for Cybersecurity (APAC) program. The details of this analyzer can be found in [15]. It is the same analyzer we have used in the later evaluation section VIII. We did a performance profiling on it to measure the total time of all set related operations when analyzing a benchmark program, Antlr, in the DaCapo [1] benchmark. The profiling result shows that more than 97% of the total analyzing time has been spent on the set related operations.

The above theoretical analysis and benchmark experiment both reveal that the set related relational operations used by flow analysis is a major bottleneck. In this paper, we attack this bottleneck by proposing the Gödel hashing mechanism, which can be used to encode efficient data structures for flow analysis, such as sets, maps and partial orders. The idea of Gödel hashes is inspired by Gödel's incompleteness proofs [9]. Gödel's incompleteness proofs used clever strategies for encoding complex mathematical structures as individual natural numbers. With occasional help from Cantor [4], we focus on one of the strategies Gödel used for encoding propositions, and we extend it to encode sets, maps and partial orders. We term these encodings Gödel *hashes* because they are compact and because inverting the encoding is impossible.[1] Compared to traditional hashing strategies, Gödel hashes possess six attractive properties:

1) **Gödel hashes are perfect hashes.** For an ordinary hash function, inequality of the hashes implies inequality of the original objects, but equality of hashes does not imply the equality of the original objects. For a perfect hash function, the hashes of two objects are equal if only if the objects are equal.
2) **Gödel hashes are dynamic.** Unlike other perfect hashing schemes, it is not necessary to know all of the items that may be hashed in advance in order to compute a Gödel hash. Moreover, if the probability distribution of objects to be hashed *is* known in advance, then the average Gödel hash will be minimal.
3) **Gödel hashes are structurally incremental.** Given a value and its hash, it is efficient to incrementally update the hash without recomputation when the value is extended. It is also straightforward to compute the Gödel hash of complex value from the Gödel hash of its components.
4) **Gödel hashes are compact.** Because Gödel hashes are dynamic and perfect, they are necessarily unbounded in size. Even so, it is straightforward to reason about their size in advance in both worst and average cases, and even their worst case is remarkably compact under pessimistic assumptions. For example, *in the worse case*, the Gödel hash of a set achieves a density of more than one element per 64-bit word until 2^{58} elements are in the universe. The average case is slightly more compact.
5) **Gödel hashes are efficient.** Operations on Gödel hashes are efficient, and they are easy to implement in most modern programming languages, thanks to their built-

[1] This statement is false. It is actually *computationally intractable* to invert.

in support for arbitrary-precision integer arithmetic. In particular, multiple precision arithmetic can be efficiently implemented by modern CPUs' SIMD instructions.

6) **Gödel hashes are partial-order-preserving.** Gödel hashes are order-preserving (monotonic) for a variety of partial orderings. For instance, if:

$$H : \mathbb{X} \to \mathbb{H}$$

computes the Gödel hash of an element in the set $\mathbb{X}$, and the relation $(\subseteq)$ orders $\mathbb{X}$ while the relation $(\sqsubseteq)$ orders $\mathbb{H}$, then:

$$x \subseteq y \text{ iff } H(x) \sqsubseteq H(y).$$

Critically, the relation $\sqsubseteq$ has a fast, efficient arithmetic implementation. We show that all *factorable* partial orders (defined in Section V) have a Gödel hashing scheme, and we show that factorability is preserved across many set-construction operations. Thus, even complex partially-ordered data structures have an order-preserving Gödel hash. Existing hashing techniques preserve *total* orders, so this opens up new possibilities for the use of hashes in flow analysis, as shown in Section VIII on pushdown control flow for object-oriented programs.

a) Key idea: Even though the precise algorithm for constructing a Gödel hash differs from one kind of structure to another, all Gödel hashing techniques that we discuss in detail exploit the same principle—The Fundamental Theorem of Arithmetic:

> *Every natural number has a unique decomposition as the product of prime factors.*

We find links between insertion and multiplication, between removal and division, and between subsumption and divisibility. We generalize these links to cover additional structures, such as partial orders.

Our message is that Gödel hashes are efficient and useful for applications in which hashes need to compactly preserve structure. As our motivating application, flow analysis is a representative use-case of Gödel hashes, which heavily relies on relational operations on partial order preserving flow sets.

A. Overview

We begin with a review of preliminary mathematics and notations. We first explore Gödel hashes on sets by defining a perfect hashing function that exploits the fundamental theorem of arithmetic; it maps set-theoretic operations on values (*e.g.*, union, intersection, subset-inclusion) into arithmetic operations on hashed values (*e.g.*, lcm, gcd, divisibility).

We conduct a formal analysis of the worst- and average-case space usage of the set hashes, and then provide a *space-optimality* result when the distribution of elements amongst sets is known *a priori*. Following the analysis of space usage, we analyze the speed of operations on Gödel hashes, which reveals that, asymptotically, some operations on Gödel hashes are *worse* than the equivalent operations on the original values.

However, by "unhiding" the hidden constant factors—the speed-up from word-sized arithmetic operations for the hashes and the cost of cache misses for the original values—we find the potential for operations on Gödel hashes to be significantly faster (a point later validated by our empirical trials).

With Gödel sets defined, we then extend Gödel hashes to maps, relations and graphs, which are also important data structures in classic program flow analysis. After that, we show that many recursively constructed, partially ordered data structures have order-preserving (monotonic) Gödel hashes. We do so by finding a condition on partial orders, factorability, which implies the existence of an order-preserving Gödel hash, and show that common set constructors preserve factorability.

We then briefly discuss computing the prime numbers necessary for Gödel hashing to work.

To evaluate compactness in space, we compare Gödel hashes on sets against array-backed sets, tree-backed sets, traditional hash-backed and bitmap-backed sets. In sparse set cases, we find order-of-magnitude reductions in size. We also find order-of-magnitude advantages in the critical operations for speed: equality, inclusion and competitive performance in other operations.

To wrap up, we discuss real world application of Gödel hashes in a program analyzer. We instrumented Gödel hashes in a real-world analysis for object-oriented programs. For the DaCapo benchmarks, analysis with Gödel hashes runs tens of times faster than the one based on original data structures.

II. Preliminaries: Background and notations

The set $\mathbb{N} = \{0, 1, 2 \ldots\}$ is the set of natural numbers. The set $\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$ is the set of integers. The set $\mathbb{P} = \{2, 3, \ldots\}$ is the set of prime numbers. The number p_i is the ith smallest prime number, where $p_0 = 2$.

To enhance readability and notational symmetry, we use the operations greatest common divisor ($\gcd$) and least common multiple (lcm) in infix form. There are many equivalent definitions of these operations, but this work exploits a particular (and uncommon) interpretation of these operations: as functions which minimize or maximize the exponents of two natural numbers in a factor-wise fashion; the greatest common divisor of two natural numbers $n = p_0^{m_0} \cdots p_n^{m_n}$ and $n' = p_0^{m'_0} \cdots p_n^{m'_n}$ is:

$$n \gcd n' = p_0^{\min(m_0, m'_0)} \cdots p_n^{\min(m_n, m'_n)},$$

and their least common multiple is:

$$n \operatorname{lcm} n' = p_0^{\max(m_0, m'_0)} \cdots p_n^{\max(m_n, m'_n)}.$$

We also use the divisibility relation:

$$a \mid b \text{ iff } b \bmod a = 0.$$

Our proofs often employ characteristic functions. The characteristic function of a set A is the function $\chi_A : A \to \{0, 1\}$:

$$\chi_A(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A. \end{cases}$$

We use the bar brackets to denote set cardinality: $|A|$ is the cardinality of the set A.

The function $\ln$ denotes the natural logarithm ($\log_e$) and the function $\lg$ denotes the binary logarithm ($\log_2$).

We make use of Cantor's bijection, $C^* : \mathbb{Z}^* \to \mathbb{N}$. There are many such bijections (but Cantor first proved their existence [4]). Any such bijection will work. For example, for the simple case of assigning pairs of naturals to a unique natural, the function $C_2^+ : \mathbb{N}^2 \to \mathbb{N}$ works:

$$C_2^+(x, y) = \sum_{i=0}^{x} i + \sum_{j=x+2}^{y+2} j$$

A. Gödel hashing notations

For each hashable structure X that we study, we will define a Gödel-hashing function $G_X : X \to \mathbb{N}$, so that $G_X(x)$ is the Gödel encoding of the value x. To unclutter notation, we will often shorthand $G_X(x)$ as $\|x\|_X$ or just $\|x\|$ when it is clear what X is.

III. Sets

Sets—unordered collections of elements—abound in functional (and non-functional) programming. For set-intensive applications, performance hinges on two factors (1) the space-efficiency of the underlying data structure and (2) the time-efficiency of operations on such structures: membership testing, inclusion testing, intersection, insertion, union, deletion and difference. A Gödel strategy for encoding sets delivers pragmatic efficiency in both dimensions.

We devote more details to the study of Gödel hashes of sets, because many of the results on sets (*e.g.*, correctness, efficiency, optimality) generalize to other data structures.

To construct the Gödel encoding of a set, first assume that every potential element has been assigned a unique prime number; then compute the product of the primes assigned to each element; the result is the Gödel hash of the set. It is not necessary to pre-construct the assignment from elements to primes: new elements may be assigned fresh primes as they are encountered for the first time.

b) Example: If the potential elements of a set are A, B, C and D, then we can assign these elements the primes 2, 3, 5 and 7 respectively. Thus, the Gödel hash of the set $S = \{A, C\}$ is the natural number $2 \times 5 = 10$. □

On the Gödel hash of a set, familiar number-theoretic operations become set-theoretic operations: modulo tests both membership *and* subset-inclusion; union becomes least common multiple; and intersection becomes greatest-common divisor.

A. Formal definition of Gödel encoding for sets

A **universe of discourse**, denoted $\mathbb{U}$, which may be either finite or infinite, is a collection of all the elements that may appear in a set. A **prime map** for a universe $\mathbb{U}$ is an injective function $P_{\mathbb{U}} : \mathbb{U} \to \mathbb{P}$ which maps every element in the universe $\mathbb{U}$ to a unique prime number. In practice, the implementation may assign primes dynamically while memoizing them; or if the elements of the universe themselves have a perfect hash map, $H : \mathbb{U} \to \mathbb{N}$,[2] then a purely functional prime map may be used:

$$P_{\text{pure}}(u) = p_{H(u)},$$

which lends itself to a recursive strategy for constructing prime maps.

Definition III.1. *The function $G_{\mathcal{P}(\mathbb{U})} : \mathcal{P}(\mathbb{U}) \to \mathbb{N}$ computes the **Gödel hash of a set**:*

$$G_{\mathcal{P}(\mathbb{U})}\{u_1, \ldots, u_n\} = \|\{u_1, \ldots, u_n\}\| = P(u_1) \times \cdots \times P(u_n).$$

Equivalently, the Gödel hash of a set may also be defined through its characteristic function:

$$G_{\mathcal{P}(\mathbb{U})}(A) = \prod_{u \in \mathbb{U}} P(u)^{\chi_A(u)}.$$

(When only one universe is under consideration, the subscripts may be left off.)

B. Set-theoretic operations and relations

We can construct the standard set-theoretic operations and relations out of arithmetic.

Lemma III.1. *Union reduces to least common multiple:*

$$\|A \cup B\| = \|A\| \operatorname{lcm} \|B\|.$$

Proof. Let $A, B \subseteq \mathbb{U}$.

$$\begin{aligned}
\|A \cup B\| &= \prod_{u \in \mathbb{U}} P(u)^{\chi_{A \cup B}(u)} \\
&= \prod_{u \in \mathbb{U}} P(u)^{\max(\chi_A(u), \chi_B(u))} \\
&= \operatorname{lcm}\left(\prod_{u \in \mathbb{U}} P(u)^{\chi_A(u)}, \prod_{u \in \mathbb{U}} P(u)^{\chi_B(u)}\right) \\
&= \|A\| \operatorname{lcm} \|B\|.
\end{aligned}$$

□

Lemma III.2. *Intersection reduces to greatest common divisor:*

$$\|A \cap B\| = \|A\| \gcd \|B\|.$$

Proof. By argument analogous to the previous proof. □

Lemma III.3. *Set difference reduces to division:*

$$\|A - B\| = \frac{\|A\|}{\|A\| \gcd \|B\|}.$$

Proof. By extension of the previous result. □

Lemma III.4. *Membership reduces to divisibility:*

$$u \in A \text{ iff } P(u) \mid G(A)$$

[2] Created, perhaps, by Gödel hashes in the internal structure of elements.

Proof.

$$\begin{aligned} u \in A \text{ iff } & 1 = \chi_A(u) \\ \text{iff } & P(u) = P(u)^{\chi_A(u)} \\ \text{iff } & P(u) \mid P(u)^{\chi_A(u)} \\ \text{iff } & P(u) \mid \prod_{u \in \mathbb{U}} P(u)^{\chi_A(u)} \\ \text{iff } & P(u) \mid G(A). \end{aligned}$$

□

Lemma III.5. *Inclusion reduces to divisibility:*

$$A \subseteq B \textit{ iff } G(A) \mid G(B).$$

Proof. By an argument similar to the proof for membership. □

Lemma III.6. *Insertion reduces to divisibility and multiplication:*

$$\|A \cup \{u\}\| = \begin{cases} \|A\| & P(u) \mid \|A\| \\ \|A\| \times P(u) & \textit{otherwise.} \end{cases}$$

Proof. By cases in $u \in A$, $u \notin A$. □

Lemma III.7. *Deletion reduces to divisibility and division:*

$$\|A - \{u\}\| = \begin{cases} \|A\|/P(u) & P(u) \mid \|A\| \\ \|A\| & \textit{otherwise.} \end{cases}$$

Proof. By cases in $u \in A$, $u \notin A$. □

C. Space-efficiency of Gödel hashes on sets

We can derive upper bounds on the size of a Gödel hash for a set. Let U be the cardinality of the universe, $U = |\mathbb{U}|$. Using the prime number theorem, we can approximate the value of the prime p_U:

$$p_U \approx U \ln(U)$$

From this, we can approximate the number of bits required to represent p_U:

$$E_{\mathbb{U}} = \textit{size}(p_U) \leq \lceil \lg(U \ln(U)) \rceil$$

When an n-bit number and an m-bit number are multiplied, the result is an (at most) $(n+m)$-bit number. From this, we can bound the bit size of a hash with k elements:

$$\textit{size}(\|\{u_1, \ldots, u_k\}\|) \leq k \times \lceil \lg(U \ln(U)) \rceil.$$

One immediate observation, is that on 32-bit hardware, until the universe of discourse exceeds a cardinality of 193,635,250 (roughly 2^{27}), each word will hold more than one element. On 64-bit hardware, the equivalent threshold for the universe of discourse is a cardinality of 415,828,534,307,634,000 (roughly 2^{58}).

We can also predict the minimum number of elements which will fit in a single word of W bits as a function of the size of the universe:

$$k_{\min}(U) = \frac{W}{\lceil \lg(U \ln(U)) \rceil}.$$

c) Example: For example, with a universe of 2^{10} elements, a 64-bit machine would be able to roughly fit 5 elements in each word. Translated to a more concrete example, if 0CFA [23] were to compute the flow sets (where each flow set contains the lambda terms that might flow to an expression) for a program with a thousand functions, most flow sets (which tend to be very sparse) would fit in single word—a single register. Using the bitmap formulation, each flow set would consume 16 words of contiguous memory; using the traditional bucket-based hash set, each flow set would consume about 10 words; and a balanced-tree sorted set implementation would consume roughly 15 words of (non-contiguous) memory per flow set. □

The average case is slightly better. For the average case, we assume that the elements of a set are uniformly distributed throughout the universe. In this case, the average size (in bits) of a set with k elements will be:

$$\frac{k}{(U-1)} \sum_{i=2}^{U} \lceil \lg(i \ln(i)) \rceil.$$

d) Example: For instance, on average, on 64-bit hardware, each word will hold a little over 5 elements on average assuming a universe with 2^{10} elements. Or, in a universe with 2^{16} elements, each word will now hold about 3 elements on average. □

1) Optimizing the prime map for space usage: If we know *a priori* the probability distribution of elements in the universe, then we can assign primes so as to minimize the number of bits per set. If the probability of an element u appearing in a set is $f(u)$, then we can construct the vector $\vec{u}^\star \in \mathbb{U}^*$ in which elements are sorted according to decreasing frequency:

$$f(u_i^\star) \geq f(u_{i+1}^\star).$$

The optimal prime map is $P^\star : \mathbb{U} \to \mathbb{P}$:

$$P^\star(u_i^\star) = p_i.$$

The expected size (in bits) of a random set is:

$$\sum_{u \in \mathbb{U}} f(u) \lceil \lg(P^\star(u)) \rceil,$$

which leads to a space-optimality result:

Theorem III.1 (Space optimality)**.** *The prime map $P^\star$ minimizes the expected bit-size of a random set.*

Proof. Straightforward. (By contradiction.) □

a) Example: If the universe has infinite size, but its elements are distributed according to a geometric distribution with parameter $r = 1/2$, then the average set size will be roughly six bits. (Intuition: Half of all the elements will be 2, which adds only one bit to a set; a quarter of all elements will be 3, which adds only two bits; an eighth of all elements will be 5, *etc.*) □

D. Time efficiency of operations on Gödel hashes for sets

As it turns out, some operations on Gödel hashes have slightly worse asymptotic complexity than other data structures for sets. We'll need to perform a more detailed accounting of their cost with respect to modern hardware, chiefly with respect to the CPU word size, to unearth their pragmatics. To make the constant factors stand out, we'll assume 64-bit hardware. We'll also assume that the underlying implementation of arbitrary-precision natural number is an array of unsigned integers (64-bit words).

We discuss the cost of operations on two sets A and B. We assume the universe contains no more than 2^{58} elements so that a single element hash can fit into a machine word. Such a universe size is sufficient in practice. Let m be the sum of the cardinality of these sets: $m = |A| + |B|$. We will refer to the maximum number of bits in the Gödel hashes of these sets: $n = mE_{\mathbb{U}}$.

- Intersection needs to compute greatest common divisor. The Euclidian algorithm requires up to $n/64$ modulo operations on two multiple precision naturals (the Gödel hashes of two sets respectively), each of which has $O(n^2)$ time complexity. Thus, the complexity of intersection is cubic: $O(n^3)$. However, the modulo operation is performed in chunks of the word-size, which means the $O(n^2)$ complexity has a hidden $1/64^2$ constant speedup factor. Considering the other $1/64$ constant in the number of times of the modulo operations, the cubic intersection complexity actually has a $1/64^3$ constant speedup factor! On a pipelined 64-bit CPU, the back-of-the-envelope cost of the computation is up to $\left\lceil \frac{n^3}{64^3} \right\rceil = \left\lceil \frac{n^3}{262144} \right\rceil$ clock cycles!
- The cost for union is the same as intersection.
- The cost for set difference is the same as intersection, plus a division operation with two multiple precision numbers.
- The cost for element insertion is a modulo operation and a multiplication operation, both of which operate on a multiple precision number (the Gödel hash of a set) with a word-size (the Gödel hash of an element) divisor or multiplicand, so the total time complexity is $O(n)$, with a $1/64$ constant speedup factor.
- The cost for element deletion is: a modulo and a division, both of which operate on a multiple precision number (the Gödel hash of a set) with a word-size (the Gödel hash of an element) divisor. Same as insertion, it is $O(n)$ time complexity with a $1/64$ constant speedup factor.
- The complexity of membership-testing is quite efficient too: one modulo operation on a multiple precision number (the Gödel hash of a set) with a word-size (the Gödel hash of an element) divisor, which costs $O(n)$ time with a $1/64$ constant speedup factor.
- The cost of subset inclusion testing is a modulo operation with two multiple precision numbers (the Gödel hashes of two sets respectively), which is $O(n^2)$ time complexity, with a $1/4096$ constant speedup factor.
- The complexity of set enumeration is equivalent to integer factorization, which is believed to be intractable[3].

In practice, multiple precision arithmetic can be further accelerated via SIMD instructions such as SSE and AVX [12]. These instructions can operate on 256 or even 512 bits data with a single instruction. Hence, all the operations above can benefit from a much larger constant factor. What's more, compared with other common set implementations such as tree-based or bucket-based hash sets, the natural implementation of Gödel hashes as a small array of unsigned integers, which can minimize cache misses, is better-suited to modern hardware for cache efficiency. The empirical evaluation in Section VII-B on the GNU GMP based Gödel hashes validates both of these.

IV. Maps, relations and graphs

The strategy for Gödel-hashing maps, relations and graphs are derivatives of the strategy for Gödel-hashing sets.

A. Hashing maps

Finite maps can be encoded as sets of pairs; thus:

Definition IV.1. *The **Gödel hash of a map** $f : X \to Y$ with respect to prime map $P_{X\times Y} : X \times Y \to \mathbb{P}$ is $G(f)$, where:*

$$G(f) = \prod_{x \in dom(f)} P_{X\times Y}(x, f(x)).$$

B. Hashing relations

Relations can also be encoded as sets of pairs; thus:

Definition IV.2. *The **Gödel hash of a relation** $R \subseteq X \times Y$ with respect to prime map $P_{X\times Y} : X \times Y \to \mathbb{P}$ is $G(R)$, where:*

$$G(R) = \prod_{x\ R\ y} P_{X\times Y}(x, y).$$

C. Hashing graphs

A directed graph (V, E) is just a set of vertexes and a set of edges.

Definition IV.3. *Given two Gödel set-hashing functions:*

$$G_{\mathcal{P}(V)} : \mathcal{P}(V) \to \mathbb{N} \ \text{, and} \quad G_{\mathcal{P}(E)} : \mathcal{P}(E) \to \mathbb{N},$$

*the **Gödel hash of a graph** (V, E) is $G(V, E)$:*

$$G(V, E) = (G_{\mathcal{P}(V)}(V), G_{\mathcal{P}(E)}(E)).$$

If one needs a natural number instead of a pair of natural numbers, then one can apply Cantor's bijection $C : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ (used in proving the countability of the rationals) to the result [4].

[3]Except that primes are sufficiently small in most cases.

V. Partial orders

The ability of Gödel hashes to accelerate testing for subsumption under a partial order, *i.e.*, whether $x \sqsubseteq y$, is perhaps their greatest strength. Partially ordered sets (posets) play an important role in fields such as static analysis and artificial intelligence. In static analysis in particular, subsumption testing in large, complex lattices can easily consume the bulk of the runtime for an analysis. (For experimental results in accelerating static analysis with Gödel hashing, see Section VIII.)

We are able to provide a structurally recursive condition for when a partially ordered set has an order-perserving Gödel hash. Specifically, given a poset $(S, \sqsubseteq)$, we can formulate Gödel hash analogs of join ($\sqcup$), meet ($\sqcap$) and subsumption ($\sqsubseteq$) if the poset S has a *prime basis*.A poset has a prime basis if every (non-bottom) element has a unique decomposition as the least upper bound of a finite number of basis elements.

Definition V.1. *For a poset $(S, \sqsubseteq)$, the set $B \subseteq S$ is a **prime basis** if:*

$$C_1 \subseteq B \text{ and } C_2 \subseteq B,$$

and

$$\bigsqcup C_1 = \bigsqcup C_2,$$

implies

$$C_1 = C_2\text{; and}$$

for any element $s \neq \bot \in S$, there exists a set $C \subseteq B$ such that:

$$s = \bigsqcup C.$$

We will call a partially ordered set that has a prime basis a **factorable** poset. A factorable poset doesn't need to have a weakest element, but if it does, the weakest element (denoted $\bot$) is not in the prime basis. (This is analogous to excluding one from the set of prime numbers.)

Definition V.2. *The function $G_S : S \to \mathbb{N}$ computes the **order-preserving Gödel hash of a partially ordered set** $(S, \sqsubseteq)$ with prime basis B under the prime map $P_B : B \to \mathbb{P}$:*

$$G_S\left(b_1 \sqcup \cdots \sqcup b_n\right) = P_B(b_1) \times \cdots \times P_B(b_n).$$

A. Operations on factorable partial orders

As with previous Gödel hashes, common operations and relations reduce to natural arithmetic.

Lemma V.1. *Join reduces to least common multiple:*

$$\|s_1 \sqcup s_2\| = \|s_1\| \operatorname{lcm} \|s_2\|.$$

Proof. Factorability allows us to construct "characteristic functions" on the prime basis of partial orders, where $\chi_s : B \to \{0, 1\}$:

$$\chi_s(b) = \begin{cases} 1 & b \sqsubseteq s \\ 0 & b \not\sqsubseteq s. \end{cases}$$

Clearly, $\chi_{s_1 \sqcup s_2}(b) = \max(\chi_{s_1}(b), \chi_{s_2}(b))$. Now, the proof is analogous for union over sets. □

Lemma V.2. *Meet operation reduces to greatest common divisor:*

$$\|s_1 \sqcap s_2\| = \|s_1\| \gcd \|s_2\|.$$

Proof. By an argument similar to the previous proof. □

Lemma V.3. *Subsumption reduces to divisibility:*

$$s_1 \sqsubseteq s_2 \text{ iff } G(s_1) \mid G(s_2).$$

Proof. By an argument analogous to the subset-inclusion test for Gödel hashes on sets. □

b) Warning: Our definition of prime basis does not ensure that a partially ordered set defines a join (nor a meet) for any two elements. As a result, there are cases where $s_1 \sqcup s_2$ will not exist, but of course, the reduction to Gödel hashes will still assign it a number, and there is no way for the Gödel hash to know that this element does not exist in the partial order. Thus, the Gödel hash reduction for partial orders is only sound under join for join-semilattices, and under meet for meet-semilattices. It is therefore advisable to promote a partial order to a lattice before working with its Gödel hash encoding to ensure soundness. (This is not an issue for application domains such as static analysis.)

B. Recursively constructed factorable posets

We can show that the standard set-construction operators preserve factorability.

c) Factorable flat orders: A poset $(S, \sqsubseteq)$ whose order is flat is trivially factorable: its prime basis is the set S.

d) Factorable power sets: A partially ordered power set $(\mathcal{P}(S), \subseteq)$ where the order is inclusion is easily factorable: its prime basis, $B_{\mathcal{P}(S)}$, consists of the singleton sets over S: $B_{\mathcal{P}(S)} = \{\{s\} : s \in S\}$

e) Products of factorable posets: The Cartesian product of factorable partial orders is itself factorable under its product ordering. Let $(A_1, \sqsubseteq_1)$ and $(A_2, \sqsubseteq_2)$ be factorable posets with prime bases B_1 and B_2 respectively. Then the poset $(A_1 \times A_2, \sqsubseteq_{A_1 \times A_2})$ is defined so that:

$$(a_1, a_2) \sqsubseteq_{A_1 \times A_2} (a_1', a_2') \text{ iff } a_1 \sqsubseteq_1 a_1' \text{ and } a_2 \sqsubseteq_2 a_2'.$$

The prime basis for the product, $B_{A_1 \times A_2}$, is the product of the prime bases: $B_{A_1 \times A_2} = B_1 \times B_2$.

f) Disjoint unions of factorable posets: If two posets $(A_1, \sqsubseteq_1)$ and $(A_2, \sqsubseteq_2)$ have prime bases B_1 and B_2, then the prime basis for the natural ordering of the disjoint sum $A_1 + A_2$ is the disjoint sum of the prime bases: $B_{A_1 + A_2} = B_1 + B_2$.

g) Function spaces into factorable partial orders: The natural partial ordering of functions leads to a factorable space of partially ordered functions when the range of the function space is factorable. That is, if $(Y, \sqsubseteq_Y)$ has prime basis B_y, then a space of *finite* functions $(X \to Y, \sqsubseteq_{X \to Y})$ is also factorable under the natural ordering:

$$f \sqsubseteq_{X \to Y} g \text{ iff for each } x \in dom(f) : f(x) \sqsubseteq_Y g(x).$$

The prime basis for this function space, $B_{X \to Y}$, is the set of functions that map just one element of x

into a prime basis element of the set Y: $B_{X\rightharpoonup Y} = \{\bot_{X\rightharpoonup Y}[x \mapsto b] : x \in X, b \in B_Y\}$. where the bottom function $\bot_{X\rightharpoonup Y}$ maps every element to the bottom of Y: $\bot_{X\rightharpoonup Y}(x) = \bot_Y$. unless the poset Y has no bottom, in which case $\bot_{X\rightharpoonup Y}$ is the everywhere undefined function: $\lambda x.undefined$.
Partial function spaces are identical except that the prime basis elements don't extend the bottom map:

$$B_{X\rightharpoonup Y} = \{[x \mapsto b] : x \in X, b \in B_Y\}.$$

C. Function spaces into countable total orders

While factorability is a sufficient condition for Gödel hashing, there are partial orders which are not factorable, yet which have an order-preserving Gödel hash. An important instance of this is a function space that maps into a countable total order. (In static analysis, such posets are used for must-alias and environment analysis [18], [16], [19].) If $(Y, \leq)$ is a totally ordered set, then the space of *finite*, partial functions $X \rightharpoonup Y$ has the natural partial order $(\sqsubseteq_{X\rightharpoonup Y})$: $f \sqsubseteq_{X\rightharpoonup Y} g$ iff $f(x) \leq g(x)$ for all $x \in X$.

Because Y is a countable total order, there exists an order-preserving measure function $M : Y \to \mathbb{N}$.

Definition V.3. *Given a totally ordered set $(Y, \leq)$, the **order-preserving Gödel hash of the function** $f : X \rightharpoonup Y$ under the prime map $P : X \to \mathbb{P}$ is computed by the function $G : (X \rightharpoonup Y) \to \mathbb{N}$:*

$$G(f) = \|f\| = \prod_{x \in dom(f)} P(x)^{M(f(x))}.$$

Under this definition, we have the usual reductions:

Lemma V.4. *Join reduces to least common multiple:*

$$\|f \sqcup g\| = \|f\| \operatorname{lcm} \|g\|,$$

Proof. The argument proceeds by constructing a multiset-like characteristic function $\chi_f : X \to \mathbb{N}$:

$$\chi_f(x) = M(f(x)).$$

The rest of the argument is analogous to union on multisets. □

Lemma V.5. *Meet reduces to greatest common divisor:*

$$\|f \sqcap g\| = \|f\| \operatorname{gcd} \|g\|,$$

Proof. By an argument similar to the previous proof. □

Lemma V.6. *Subsumption reduces to divisibility:*

$$f \sqsubseteq g \text{ iff } G(f) \mid G(g).$$

Proof. By an argument analogous to that of subset-inclusion testing for multisets. □

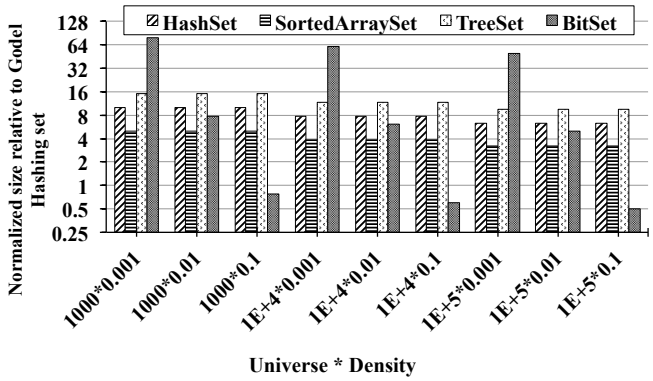

Fig. 1. Normalized size of traditional hash sets, sorted-array sets, sorted-balanced tree sets and bitmap sets, relative to the size of the predicted worst case of Gödel hashing sets. The (logarithmic scale) vertical axis is the normalized size. The horizontal axis is denoted as $U * \rho$: U is the size of the universe, ρ is the density of the set as a fraction of the universe. *The worst case of Gödel hash dominates for compactness—by up to tens of times smaller than that of the common data structures.*

VI. Computing primes and prime maps

Gödel hashes rely on being able to generate prime maps. Constructing efficient prime maps requires an efficient method for generating the ith prime number. In functional programming, a global, lazy, internally memoizing stream of prime numbers is convenient, particularly when multiple structures in the program will require their own prime maps. This is the approach that our implementation use (in the analyzer implementation in Section VIII). There are pragmatic methods that use a sieve to generate primes deterministically [21]. However, probabilistic primality tests [25], [20], [22] are more efficient, arbitrarily reliable and require no storage of prior primes. The probabilistic tests are particularly fast on word-sized primes. And, if primes are allocated in an on-demand fashion, word-sized primes are all that are likely to be needed.

VII. Experimental results

There are two key questions to answer with experimentation:
(1) How big do Gödel hashes get?
(2) How fast are operations on Gödel hashes?

The short answer to the question of size is that they are roughly tens of times less than the size of the structure from whence they came for relatively sparse data. With respect to speed, the short answer is that the critical operations of inclusion and equality are orders of magnitude faster at all densities.

We'll focus our efforts on Gödel hashes of sets, since these form the basis for the other techniques. Let the set $\mathbb{U}$ be the universe and let $U = |\mathbb{U}|$ be the size of the universe.

A. Measuring hash size

Figure 1 renders the normalized sizes of the following standard data structures, relative to that of the predicted worst-case for Gödel hashes set: (1) the array-backed set; (2) the tree-backed set; (3) the traditional hash-backed set; and (4)

the bitmap-backed set. The varied parameters are (1) the size of the universe, U, and (2) the density of the set relative to the size of the universe, ρ. For instance, with a density of 0.1 and a universe of 10,000 elements, the set size under consideration is $0.1 \times 10,000 = 1,000$.

Note that we compute the exact array size, without consideration of resizing strategy, which is commonly employed in most program languages' standard libraries for better performance (e.g. Java's `ArrayList` increases its capacity by a factor of 1.5). Also, we only compute the elements' size based on a very space-efficient bucket-based hashset implementation (C++'s `unordered_set`), the bucket array's size is ignored.

Even so, Gödel hashes are substantially smaller (tens of times smaller) than all of the other data structures, except the dense bitmap-backed sets with $\rho = 0.1$. Note that even though the bitmap set is very space-efficient for representing dense data, when it becomes sparse with small ρ, the size of bitmap set can be up to 78.4 times larger than that of Gödel hashes!

We focus on ***sparse sets*** because higher-order program analysis tends to deal with highly sparse flow sets. For instance, in context-sensitive analysis, bitmap-backed sets could be exponential in the size of the program, whereas the median flow set in practice has size two.

B. Measuring speed

In Table I, we measured the slow down ratio of average run time of each single set operation on sorted tree sets, sorted array sets, and bitmap sets relative to that of Gödel hashes. For hashes, fast equality is a critical operation. And, for Gödel hashes, fast subsumption (subset) is also critical for candidate applications. Table I shows that the critical hash operations on Gödel hashes of sets can be up to hundreds of times faster! Remarkably, the performance of critical operations on bitmap sets degrades rapidly (by an order of magnitude) when the sets become sparse (when ρ decreases), even though they can be relatively more efficient than the other three standard data structures in dense sets (with $\rho = 0.1$). The performance advantage of Gödel hashing sets on sparse sets (in addition to the size advantage validated in Table 1) fits extremely well over other data structures in the case of program analysis, especially for higher-order programs.

h) Implementation details: Gödel hashes are implemented in C++ using GNU GMP (GNU Multiple Precision Arithmetic library) for big integer arithmetic operations. GMP is highly optimized on modern CPUs to operate at very long data types with a single instruction. Instruction sets such as SSE and AVX can do 256 bits or even 512 bits data arithmetic operations [12]. Sorted tree set uses C++ `std::set` , which is red-black tree based. Sorted array set uses C++ `std::vector` to store data, and uses `std`'s binary search and set algorithms for correspondent set operations. Hash set uses `std::unordered_set` that is a bucket-based hash set. Bit set uses C++ `std::bitset`, which is implemented with an array of integers. The evaluation program runs each operation 1,000 times on sets that contains $U * \rho$ number of elements. These elements are randomly fetched from the prime universe U. The evaluation is conducted on a PC with a six-core Xeon 3.3GHz CPU and 32GB RAM.

VIII. Application: Pushdown control flow analysis for object-oriented programs

To further demonstrate the high promise of Gödel hashes, we adapt a publicly available analyzer that is developed for DARPA's Automated Program Analysis for Cybersecurity (APAC) program. It is an effective semantic-based abstract interpretation framework to detect maliciousness in Android applications [15]. However, the analysis still suffers from the performance bottleneck originating from the subsumption testing during fixed point computation. This makes it an excellent candidate to validate the benefits of Gödel hashes in improving analysis run time.

To make the evaluation more compelling, we evaluate the Gödel hashing instrumented analyzer on the widely used standard benchmark suite—DaCapo [1]. It has much larger scale of code bases to analyze than ordinary mobile applications presented in Google market. What's more, the realistic workload can stress-test analysis with Gödel hashing substantially.

Since the analyzer works directly on Dalvik byte code, which is compiled from Java programs in Dalvik Virtual Machine (DVM), we have successfully compiled 10 out of 11 Java applications in the DaCapo benchmark (v.2006-10.MR2) in DVM with minor change in the source code (mainly changed `enum` to other names because `enum` is a key word in JRE 1.5 or later). We encapsulate the `main` method of each benchmark in the entry point `onCreate` of a class of type `Activity`. These benchmarks are compiled using the built-in tool `dx` in Android SDK. Some GUI class references (especially `awt`) in Java programs are resolved by including `rt.jar` in the Android class path. To avoid name space conflicts in packages/classes, we use `jarjar` [13] to repackage some Java standard libraries that are re-implemented in Android. The only Java program that is not ported is `eclipse`, which involves substantial conflicts in Java GUI class (`awt, swing, swt`). We believe the other ten available programs suffice for our purpose.

Table II presents the runtime speedup for the DaCapo benchmarks that are compiled in Android DVM. With Gödel hash domains, the analysis can run tens of times faster than the one without.

IX. Related Work

There is an intellectual debt in this work to Kurt Gödel [9] and Georg Cantor [4]. It is arguably true that Cantor's bijections were the first perfect hash functions.

Gödel did not actually use most of encodings we describe, but we feel justified in attributing his name to them, because it is so similar in spirit to his encodings. Gödel encoded a sequence of natural numbers $\langle n_1, \ldots, n_m \rangle$ by raising the ith prime (p_i) to n_i, and then multiplying the results together: $\langle n_1, \ldots, n_m \rangle \equiv p_1^{n_1} p_2^{n_2} \cdots p_m^{n_m}$. It is straightforward to extend this line of thinking to sets, multisets and partial orders as we have done.

TABLE I

SLOW DOWN RATIO OF AVERAGE RUN TIME OF EACH SINGLE SET OPERATION ON SORTED TREE SETS, SORTED ARRAY SETS, HASH SETS AND BITMAP SETS, RELATIVE TO THAT OF GÖDEL HASHES. *For the critical hash operations of equality and subsumption, operations on Gödel hashes of sets are up to hundreds of times faster.*

U	**density**		$\subseteq$	$=$	$\cup$	$-$	$\cap$	$\in$	deletion	insertion
5,000	0.001	sorted treeset	2.667	2.333	1.459	1.179	0.56	1	2.667	3.333
		sorted arrayset	1.333	1.333	0.324	0.393	0.36	0.75	1	2.333
		hashset	4	4.333	1.757	1.607	0.72	1	1.667	2.333
		bitset	36.333	35.333	2.216	2.929	3.32	0.25	0.333	0.333
	0.01	sorted treeset	17.75	5.667	1.067	0.888	0.372	1.333	3.2	2.25
		sorted arrayset	6.625	2.333	0.098	0.105	0.072	0.833	1.6	2.5
		hashset	14	5.333	1.318	0.82	0.287	0.667	1.2	1
		bitset	13.75	23.333	0.129	0.147	0.149	0.333	0.2	0.25
	0.1	sorted treeset	89.147	285.455	0.843	0.789	0.399	1.278	1.962	2
		sorted arrayset	40.941	112.818	0.075	0.07	0.049	0.556	1.423	4.32
		hashset	31.294	94.182	0.921	0.551	0.201	1.444	0.923	0.72
		bitset	3.235	9.727	0.01	0.011	0.011	0.111	0.038	0.08
10,000	0.001	sorted treeset	4.8	9	1.165	1.056	0.395	1.25	3.667	2.333
		sorted arrayset	1.8	3.5	0.165	0.222	0.158	0.75	1.333	2
		hashset	4	11	1.66	1.25	0.421	0.75	2	1.167
		bitset	49.4	117.5	1.641	2.319	2.197	0.5	0.667	0.333
	0.01	sorted treeset	36.8	84.25	0.952	0.797	0.349	1	2.571	3
		sorted arrayset	12	26.5	0.081	0.084	0.059	0.636	1.714	4.5
		hashset	22.2	55.5	1.168	0.645	0.235	0.273	0.857	1.333
		bitset	24.6	59	0.129	0.141	0.141	0.182	0.286	0.5
	0.1	sorted treeset	111.882	392.737	0.696	0.626	0.313	2.345	1.352	1.605
		sorted arrayset	58.853	205.737	0.052	0.048	0.034	0.448	1.759	5.698
		hashset	39.235	134.895	0.624	0.39	0.144	0.586	0.407	0.767
		bitset	3.559	12.105	0.007	0.007	0.007	0.069	0.037	0.047
50,000	0.001	sorted treeset	17.875	40	0.829	0.668	0.262	1	3	3.4
		sorted arrayset	6.5	14.667	0.077	0.08	0.051	0.714	1.6	4
		hashset	14	37.333	1.048	0.586	0.201	0.429	1.2	1.6
		bitset	247.875	599.333	1.691	1.739	1.663	0.571	1	1.2
	0.01	sorted treeset	76.244	259.333	0.62	0.576	0.241	1.091	1.455	1.769
		sorted arrayset	35.463	104.667	0.05	0.048	0.032	0.409	0.939	4.038
		hashset	27.342	91.917	0.699	0.391	0.14	0.227	0.333	0.5
		bitset	48.342	150.667	0.113	0.118	0.115	0.227	0.152	0.154
	0.1	sorted treeset	158.884	596.427	0.32	0.339	0.201	0.888	0.369	0.589
		sorted arrayset	58.382	219.281	0.018	0.018	0.012	0.231	1.65	6.386
		hashset	52.693	197.812	0.254	0.238	0.131	0.806	0.136	0.35
		bitset	5.565	19.104	0.004	0.004	0.004	0.037	0.016	0.03
100,000	0.001	sorted treeset	29.462	68.6	0.747	0.62	0.281	0.917	2.25	2.429
		sorted arrayset	8.769	20.2	0.062	0.065	0.045	0.583	1.5	3.857
		hashset	17	44	0.877	0.501	0.183	0.333	0.875	1.286
		bitset	338.077	1,001	1.795	1.868	1.865	0.5	1.125	1.143
	0.01	sorted treeset	96.747	341.455	0.483	0.464	0.189	1.457	1.045	1.059
		sorted arrayset	51.279	179.318	0.035	0.033	0.022	0.314	1.358	3.353
		hashset	35.57	123.909	0.494	0.28	0.1	0.286	0.254	0.324
		bitset	54.392	179.136	0.093	0.098	0.098	0.2	0.119	0.118
	0.1	sorted treeset	141.405	414.681	0.224	0.24	0.146	0.471	0.193	0.246
		sorted arrayset	51.9	152.039	0.012	0.013	0.008	0.143	1.54	4.825
		hashset	68.372	199.84	0.204	0.203	0.116	0.611	0.081	0.173
		bitset	5.403	14.858	0.003	0.004	0.004	0.027	0.012	0.015

TABLE II
ANALYSIS RUNTIME SPEEDUP WITH GÖDEL HASHES IN DACAPO BENCHMARKS

benchmark name	lines	speed-up
antlr	35,000	22.2x
bloat	70,344	5.9x
chart	217,788	14.2x
fop	184,316	11.7x
hsqldb	155,591	18.7x
luindex	38,221	33.4x
lusearch	87,00	30.1x
pmd	55,000	17x
xalan	259,026	14.7x

Gödel also devised a β-function encoding for sequences. An advantage of the β-function encoding, which uses the Chinese remainder theorem, is that it does not require factoring or knowledge of primes in order to retrieve the ith element of a sequence. We touched upon the β-function technique, but we reserve a fuller exploration of its own unique advantages for future work.

The largest body of related work lies in the field of hashing functions. More specifically, this work fits within a subset of a subset of a subset of a subset of the field: dynamic, order-preserving, perfect hash functions on complex data structures. Perfect hash functions are discussed in [10] and [14]. Previous work in [2], [3], [5], [6], [8], [7], [11], [26] all examine perfect hashing, but only [5] and [8] preserve order, and then, they preserve only a total order. Neither of these techniques is dynamic; that is, they require foreknowledge of the keywords to be hashed. According to our survey of the literature, we believe Gödel hashes to be the only member of their class.

X. CONCLUSION

Motivated by the flow analysis performance bottleneck caused by set related relational operations, we proposed Gödel hashes as a speedup technique in flow analysis. Inspired by Kurt Gödel's proofs of incompleteness, Gödel hashes arise from the fundamental theorem of arithmetic's guarantee of the uniqueness of integer factorization. Remarkably, prime factorization ends up being able to suitably capture important properties for sets, maps, relations, graphs and partial orders. Gödel hashes bring a unique confluence of properties to hashing: Gödel hashes are perfect and order-preserving yet still dynamic, structural and incremental; Gödel hashes are not minimal, but they are compact. In practice, Gödel hashes substantially improve performance for the critical operations of inclusion and equality. In real world benchmark experiment, Gödel hashes based static flow analysis has demonstrated tens of times speedups.

The source code, implementation, benchmarks and Gödel-hashing library used in this paper are available from: http://www.cs.utah.edu/~liangsy/godelhash-src.html.

XI. ACKNOWLEDGMENTS

This material is partially based on research sponsored by DARPA under agreement number FA8750-12-2-0106 and by NSF under CAREER grant 1350344. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

REFERENCES

[1] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of OOPSLA '06*. ACM, October 2006.

[2] M.D. Brain and A.L. Tharp. Near-perfect hashing of large word sets. *Software – Practice and Experience*, 19:967–978, 1989.

[3] M.D. Brain and A.L. Tharp. Perfect hashing using sparse matrix packing. *Information Systems*, 15:281–290, 1990.

[4] Georg Cantor. Über eine eigenschaft des inbegriffes aller reellen algebraischen zahlen. *Mathematische Annalen*, 1874.

[5] C.C. Chang. The study of an ordered minimal perfect hashing scheme. *Communications of the ACM*, 27(4):384–387, April 1984.

[6] C.C. Chang. A letter-oriented minimal perfect hashing scheme. *The Computer Journal*, 29(3):277–281, June 1986.

[7] R.J. Cichelli. Minimal perfect hash functions made simple. *Communications of the ACM*, 23(1):17–19, January 1980.

[8] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. An optimal algorithm or generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257–264, October 1992.

[9] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte fr Mathematik und Physik*, 38:173–198, 1931.

[10] G.H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1991.

[11] G. Haggard and K. Karplus. Finding minimal perfect hash functions. *ACM SIGCSE Bulliten*, 18:191–193, 1986.

[12] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 1*. Intel Corporation, September 2013.

[13] JarJar. Jarjar. https://code.google.com/p/jarjar/.

[14] T.G. Lewis and C.R. Cook. Hashing for dynamic and static internal tables. *Computer*, 21:45–56, 1988.

[15] Shuying Liang, Matthew Might, David Van Horn, Steven Lyde, Thomas Gilray, and Petey Aldous. Sound and precise malware analysis for android via pushdown reachability and entry-point saturation. In *Proceedings of the third ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '13. ACM, 2013.

[16] Matthew Might. *Environment analysis of higher-order languages*. PhD thesis, Georgia Institute of Technology, 2007.

[17] Matthew Might, Benjamin Chambers, and Olin Shivers. Model checking via ΓCFA. In *Proceedings of VMCAI'07*, pages 59–73, Jan 2007.

[18] Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of ICFP'06*, pages 13–25, Portland, Oregon, Sep 2006.

[19] Matthew Might and Olin Shivers. Exploiting reachability and cardinality in higher-order flow analysis. *Journal of Functional Programming*, 18(5–6):821–864, 2008.

[20] Gary L. Miller. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences*, 13(3):300–317, 1976.

[21] Melissa E. O'Neill. The genuine sieve of eratosthenes. *Journal of Functional Programming*, October 2008.

[22] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[23] Olin Shivers. Control-flow analysis in Scheme. In *Proceedings of PLDI'88*, pages 164–174, 1988.

[24] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages, or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[25] Robert M. Solovay and Volker Strassen. A fast monte-carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, 1977.

[26] V.G. Winters. Minimal perfect hashing in polynomial time. *BIT*, 30(2):235–244, 1990.